AN EXPLORATORY ANALYSIS OF THE .NET COMPONENT MODEL AND UNIFRAME PARADIGM USING A COLLABORATIVE APPROACH

A Thesis

Submitted to the Faculty

of

Purdue University

by

Natasha Sushil Gupta

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2004

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
AUG 2004		**00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An Exploratory Analysis of the .Net Component Model and Uniframe Paradigm Using a Collaborative Approach	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Indiana University/Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**186**	

To Mamma

ACKNOWLEDGMENTS

I would like to express my sincere gratitude towards Dr.Rajeev Raje for giving me the opportunity to work on the UniFrame project and help me complete my Masters with valuable research experience. His guidance, encouragement, support and patience has helped me through every step of my study period. Also, I would like to gratefully acknowledge the guidance and support rendered by Dr. Andrew M. Olson throughout my research period and help in successfully accomplishing this thesis. My sincere thanks to Dr. Stanley Chein for allowing me to harness this opportunity to work on the UniFrame project that has helped me grow both professionally and personally. I also appreciate Dr. Dongsoo Kim for agreeing to be a part of my graduate committee.

Whole-heartedly, I would also like to thank Jim Freeman and Joe Hansome for their cooperation and help in implementing a significant part of my prototype. I wish them good luck in their careers. Special thanks to Valerie Lim Diemer for her patient guidance in the formatting of the thesis. I would like to thank the U.S. Department of Defense and the U.S. Office of Naval Research for supporting this research under the award number N00014-01-1-0746.

Last but not least; I take this opportunity to sincerely thank the entire Department of Electrical & Computer Engineering, Department of Computer & Information Science, all my peers and friends for their cooperation and help during this important phase of my life; and I would like to extend a special thanks to my colleague, Kalpana Tummala for her immense contribution in helping me meet my thesis deadlines successfully.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF TERMS

Acronym	Term
AM	Adapter Manager
AR	Active Registry
B2B	Business To Business
CAO	Client Activated Object
CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off – The Shelf
DCOM	Distributed Component Object Model
DCS	Distributed Computing System
DM	Discovery Manager
DSM	Domain Security Manager
EAI	Enterprise Application Integration
GDM	Generative Domain Model
GIIS	Grid Index Information Service
GRIS	Grid Resource Information Service
HH	Headhunter
ICB	Internet Component Broker
J2EE	Java 2 Enterprise Edition
J2N	Java To .NET
LM	Link Manager
MDA	Model Driven Architecture
N2J	.NET To Java
QM	Query Manager
QoS	Quality of Service

Acronym	Term
RMI	Remote Method Invocation
SAO	Server Activated Object
SOAP	Simple Object Access Protocol
TLG	Two Level Grammar
UA	UniFrame Approach
UDDI	Universal Description, Discovery and Integration
UMM	Unified Meta-component Model
UQoS	UniFrame Quality of Service
URDS	UniFrame Resource Discovery Service
WS	Web Services
WSDL	Web Services Description Language
XML	EXtensible Mark-up Language

ABSTRACT

Gupta, Natasha Sushil. M.S.E.C.E., Purdue University, August 2004. An Exploratory Analysis of the .NET Component Model and UniFrame Paradigm Using a Collaborative Approach. Major Professors: Stanley Y. P. Chien and Rajeev R. Raje.

The emergence of the Distributed Computing paradigm has laid down numerous big challenges amidst the computing world. The advantages offered by a distributed system as against a centralized one makes these challenges all the more necessary to be overcomed. Currently, the computing industry is striving to achieve solutions to such questions.

Microsoft's .NET Framework has emerged as one of the computing paradigms in response to such challenges. The aim of this study is to study the framework in detail and provide a background/basis for the incorporation of this framework in the context of the project UniFrame. The study expands on different fronts overlapping both .NET and UniFrame. These aspects include the comparison of rapidly growing Web Services from the UniFrame's point of view in addition to the architectural issues incorporated in the realization of a UniFrame Resource Discovery Service on a .NET platform. The thesis also explores issues related to the interoperability of the .NET framework with other computing models and suggests an approach based on a thorough study and experimentation. This thesis clearly indicates that for a platform to truly support integration of distributed computing systems, major challenges need to be addressed such as dynamic discovery, registration and quality of service assurance and an approach is required to tackle them across heterogeneous component models.

1 INTRODUCTION

The computing world has experienced a growing shift of the computing paradigm from a centralized to a distributed one, for the past few years. More and more computers are now connected to one another so that their capabilities can be shared over the network creating the realm of distributed computing. Distributed Computing systems (DCS) group individual computers together and pool their associated computing resources in order to accomplish higher level computation in terms of compound systems. Resources, both hardware and software, may be managed by servers and accessed by clients or they may be encapsulated as objects and accessed by other client objects. The result in either case is programming infrastructure composed of numerous, scattered and autonomous work stations collaborating as a single integrated system. The Internet, intranet, and spontaneous networking, are all examples of distributed systems. DCS offer several advantages for improving availability and reliability through replication, performance through parallelism and in addition flexibility, expansion, and scalability of resources. Due to the benefits achieved with the use of DCS, distributed systems are posed to become the primary computing infrastructure for scientific work and the industry. These advantages are however associated with certain challenges inherent to the world of distributed computing [DIS01] and are described in the following paragraphs.

Heterogeneity: DCS must be constructed from variety of different networks, operating systems, computer hardware and programming languages. One possible solution is the use of the Internet communication protocols to mask the differences in networks, and there is a variety of middleware that can deal with other differences, as will be discussed in the following paragraphs.

Openness: The openness of a distributed system is the characteristic, which determines whether the system can be extended and re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Security: Information resources that are made available and maintained in the DCS may hold a high intrinsic value for the owners of the information, for example, in the domain of defense and medical care. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized users); integrity (protection against alteration and/or corruption); and availability (protection against interference with the means to access the resources).

Scalability: Distributed systems operate effectively and efficiently at many different scales, ranging from small intranet to the Internet. A DCS is described as scalable if it remains effective when there is a significant increase in the number of resources and the number of users.

Failure handling: Distributed systems are characterized by partial failures, i.e., some components of the system fail, while others continue to function normally. Hence, failure handling is of critical importance in a DCS. Every DCS hence needs to address questions such as detecting failures, masking failures, tolerating failures and recovery from failures.

Concurrency: The concept of “shared resources” in a DCS implies inherent concurrency where both services and applications can be shared by multiple clients. A DCS must be responsible to ensure that the shared resources are accessed in a safe and synchronized way in a concurrent environment.

Transparency: This challenge is defined as the concealment from the user and the application programmer of the separation and disparity of the components in a DCS, so that the system is perceived as a whole rather than a collection of independent of components.

One of the promising approaches to handle these challenges and allow the software design of DCS is based on the principles of distributed component computing (DCC). Under this paradigm, integrating geographically scattered heterogeneous software components creates DCS. These components constantly discover one another, offer/utilize services, and negotiate the cost and quality of services. Such a vision provides for a scalable solution and hides the underlying heterogeneity [RAJ01].

The principles of DCC led to the emergence of component middlewares for developing commercial off-the shelf (COTS) components. Component middleware encapsulates specific services or a set of services to provide reusable building blocks that can be composed to develop DCS more rapidly and robustly than those built from scratch. In particular, component middleware offers the following reusable capabilities [GOK02]:

- *Horizontal infrastructure services*, such as request brokers,
- *Vertical models of domain concepts*, such as common semantics for higher-level reusable component services, and
- *Communication mechanisms* between components such as remote-method invocation or message passing.

A variety of such COTS component middlewares are prevalent and widely used. Examples are Common Object Request Broker Architecture (CORBA) [COR01], Java 2 Enterprise Edition (J2EE) [JAV01], DCOM [MIC01], and .NET [MIC02]. Despite the advances in the ubiquity and quality of component middleware, the following challenges still exist in constructing DCS out of component midlewares and hence need to be addressed.

Proliferation of middleware technologies: A majority of the middleware platforms are designed for "closed" systems. However, in a DCS, there is a need for the middleware platforms to work with heterogeneous platforms and languages, interfaces with legacy code written in different languages and interoperate across multiple middleware used by different component developers. The COTS component middleware, however, do not provide a complete end-to-end solution to support DCS development in a diverse environment.

Satisfying multiple qualities of service requirements: A large number of distributed systems have stringent requirements of quality of service (QoS) demands such as efficiency, scalability, dependability, and security that must be satisfied and that require end-to-end enforcement to the whole of DCS. Conventional implementations of component middleware are unable to enforce complex QoS requirements effectively

In addition, there are myriad strategies for configuring and deploying different underlying middlewares so that it becomes a daunting task to assemble a DCS composed of incompatible COTS components. Hence, a comprehensive framework that provides seamless access to the underlying components and aids in the design of a DCS is required. UniFrame [RAJ01] is one such example of a framework that aims to construct a DCS, while using automation, out of heterogeneous components that conform to a quality of service. It provides a paradigm to systematically apply the notion of domain-specific models to engineer a DCS, while hiding the complexities associated with different COTS component middleware used to develop the components. Design of such a framework entails addressing all the above-mentioned challenges.

.NET [MIC02] is one of the prominent component middleware in the field of distributed component-based computing. An inclusion of the .NET component middleware within the UniFrame requires an exhaustive study and investigation into its underlying model. The thesis presents a synergistic approach to such a study in the context of .NET and UniFrame.

1.1 Problem Definition and Motivation

The UniFrame approach [RAJ01, RAJ02] incorporates the following key concepts: a) a meta-component model (the Unified Meta Model – UMM [RAJ00]), with a associated hierarchical setup for indicating the contracts and constraints of the components and associated queries for integrating a distributed system, b) an integration of the QoS at the individual component and distributed application levels, c) the validation and assurance of the QoS, based on the concept of event grammars, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available component choices.

The UniFrame's approach to creating a DCS out of heterogeneous components requires understanding and then overcoming the wide disparities that exist between different component models. Hence, the underlying approach, in UniFrame, is largely dependent on tackling issues that arise while bridging different component middleware.

Middleware is reusable software that resides between the applications and the underlying operating systems, network protocol stacks and hardware. Its primary goal is to bridge the gap between the application programs and the lower-level hardware and software infrastructure to coordinate how parts of the applications are connected and how they interoperate [GOK02].

Since every component model is fairly comprehensive, it contains an associated methodology and architecture that needs to be used while interoperating with other component models. Thus, the incorporation of any component model into the UniFrame requires a thorough understanding and investigation of the component model when placed in the context of the UniFrame. This study chooses one particular eminent component model, .NET, and examines its incorporation into the UniFrame. During the inclusion process, this thesis also addresses many above-mentioned interesting questions, such as, does the .NET component model solve all the problems associated with the construction of a DCS?, what changes need to be made if a component created by using

the .NET model has to interoperate with components implemented using another models?, and what are the trade-offs associated with this process?

1.2 Research Goals of the Thesis

The objective of this thesis of encompassing .NET into UniFrame is subdivided into three specific goals:

- To analyze and compare the underlying distributed computing model of the .NET paradigm with respect to the approach used by UniFrame.
- To adapt and experiment with the discovery aspect of the UniFrame in the context of .NET.
- To propose an architecture, using the UniFrame principles, for interoperating between the .NET and Java-RMI models.

1.3 Contributions

.NET has emerged as one of the widely accepted component models and provides the infrastructure to support organizing distributed components into service-oriented architectures for the realization of a DCS. However, even though the model supports inherent features for achieving interoperability necessary to achieve a DCS, this study indicates that, there are a number of issues that it still needs to consider in order to truly tackle the problem of heterogeneity. Firstly, like other component models, .NET assumes the presence of other components adhering to the same model in a DCS. This might not be the case in a large scale decentralized system and hence even though the model is complete in its own realm, it cannot truly achieve an efficient construction of a DCS to offer a variety of services. Thus, there is a need for a meta-model which can tackle the problem of heterogeneity in compliance with the principles of local autonomy for distributed component computing. Since, UniFrame provides for such a meta-model, it can overcome the limitations of the .NET component model. However, this requires the incorporation of this model into the UniFrame which is a challenging task since it

requires a thorough evaluation of the different aspects associated with .NET (or any other component model). This thesis provides for such a guideline for leveraging the .NET component model at different fronts identified for the model, such as registration and dynamic discovery of .NET components and interoperability with other component models for an efficient composition of a DCS. Since, the study is carried out in conjunction with UniFrame, this thesis also contributes to the evaluation of the UniFrame approach relative to one of the major challenge it faces, namely heterogeneity, indicating that the principles of UniFrame lead to a better approach for DCS construction than solely .NET and at the same time can encompass .NET's functionality provided it is able to tackle different technological related issues faced, an area where this thesis contributes. The deliverables of this thesis are:

- This research provides the metrics for comparing the underlying distributed models of the .NET and UniFrame and describes an empirical evaluation based on these metrics.
- It proposes and implements a platform-specific discovery architecture (.NET Remoting-based) from a known platform-independent UniFrame resource discovery service.
- It provides an approach for the federation of the URDS instances, which spans across heterogeneous discovery services. The approach is validated with a prototypical implementation and associated experimentation.

1.4 Organization

The thesis is organized into seven chapters. The first chapter provides an introduction to the overall goal of the thesis with the focus on problem statement and the motivation behind the work. This in turn followed by chapter two which describes the related and previous work. The problem of heterogeneity is discussed in the chapter three. The chapter four provides a detailed analysis of the Web Services framework of the .NET paradigm in comparison to the UniFrame approach. The chapter five describes the UniFrame Resource Discovery Service as incarnated in the .NET component model.

The chapter six studies the problem of heterogeneity in the context of UniFrame and .NET and provides for experimental validation in terms of federation of heterogeneous discovery services and interoperation studied with respect to .NET. Finally, the chapter seven concludes the thesis with the summary of the study and future work.

2 RELATED AND PREVIOUS WORK

In reference to the three main sub-goals of the thesis, namely the analysis of the UniFrame and Web Services paradigms, adaptation of the URDS architecture in the context of .NET and studying interoperability with respect to the interconnection of discovery services; this chapter provides an introduction to some of the works related to this study. Section 2.1 introduces the UniFrame paradigm which forms an integral part of the previous work on which this study is based. Section 2.2 then introduces the .NET component model in brief. Other details of the framework are incorporated in later chapters as and when needed. The next step that the thesis takes in order to study the collaboration of the UniFrame model with the .NET framework is the mapping of the URDS architecture to a .NET-based implementation. The reconstruction is also studied in terms of performance and heterogeneity in Chapters 5 and 6 respectively. Hence, it becomes important to discuss some of the related work in this direction.

Section 2.3 briefly describes the different works in this direction. Encompassing the .NET component model into the UniFrame also entails certain interoperability issues with respect to other component models at different areas of the UniFrame. The study includes experimentation with this aspect in terms of interoperability between the .NET component model and the Java RMI model. Section 2.4 discusses the interoperability provided in this direction by some of the commercial bridges that were studied and experimented with as part of the study.

2.1 Introduction to UniFrame

The main objective of UniFrame research initiative is to provide a framework for the seamless integration of distributed heterogeneous components. UniFrame aims at achieving automation (to the extent possible) while integrating these components and also stresses the quality assurance of individual components and the system made out of them. The UniFrame Approach provides the overall process and is based on the Unified Meta-component model. More details about these can be found in [RAJ00], [RAJ01] and [RAJ02]. The next two sections discuss the process and incorporate various aspects such as UMM specifications, the UniFrame Quality of Service Framework and the UniFrame Resource Discovery Service (URDS) in brief; the URDS will be discussed in detail in Chapter 5. The details are based on the work from above references.

2.1.1 Unified Meta-component Model (UMM)

In providing the overview of the UniFrame process the first step is to introduce the UMM which is the foundation of the approach. The recent shift in the focus of the Object Management Group (OMG) to Model Driven Architecture (MDA) [OMG01a] is a recognition that bridging components to create a distributed computing system requires standardization of not only the infrastructure but also Business and Component models. The UMM is one such standard aimed at providing a unifying meta-model for the purpose of enabling discovery, interoperability, and collaboration of components using the generative techniques. It consists of the following main parts: components, services and service guarantees, and infrastructure.

Components: In UMM, components are autonomous entities that have well-defined interfaces and private implementations. Any communication with a component is through its interfaces which act as the input and the output channels for the component. The components may adhere to diverse distributed computing models. In addition, each component in UMM is defined by three main aspects: a) computational aspect, b) cooperative aspect and c) auxiliary aspect. The computational aspect of a component

enables UMM components to support the notion of “introspection” by which it precisely describes its functionality to other components in a DCS.

Service and Service Guarantees: The service aspect of each component is the means by which a component is able to specify the quality of the service it offers. This aspect is significant in a DCS where there could be multiple choices for a particular component. The quality assurance of a service provided by the developer enables a system integrator to have some form of guaranty about the performance of the service during its deployment and utilization in a bigger system. Some of the factors identified in the UMM, based on which the QoS of a component could be measured are algorithm used, its expected computational effort, required resources, and etc. UniFrame is a quality-oriented framework and the UMM provides for a UniFrame Quality of Service Framework (UQoS) [BRA01] to guaranty the necessary QoS, both at the component and system’s level.

Infrastructure: The UMM provides a discovery infrastructure which enables the discovery of components belonging to the different component models and based on certain functional and non-functional constraints. The infrastructure is primarily made up of the concepts of the headhunter and Internet Component Broker [RAJ02]. Some of these aspects of the UMM will be further discussed in detail in the coming sections under Chapter 3 when describing these aspects in detail with respect to the Web Services Framework.

2.1.2 UniFrame Approach (UA)

The UniFrame process of composing distributed computing system can be explained with the help of Figure 2.1 (with graphics from due permissions from [CLI02]). The process proceeds in the following two main parts [RAJ02],

- Component development and deployment.
- Automated system generation and its QoS-based evaluation.

Component Development and Deployment: This phase is built on the notion that the UniFrame approach is based on the Generative Programming [CZA00] paradigm. Therefore the process has an underlying assumption that the generative environment of the UniFrame is constructed around a generative domain specific knowledge (GDM) supporting component-based assembly. This implies that the components are created for a specific application domain, based on an accepted and a standardized GDM.

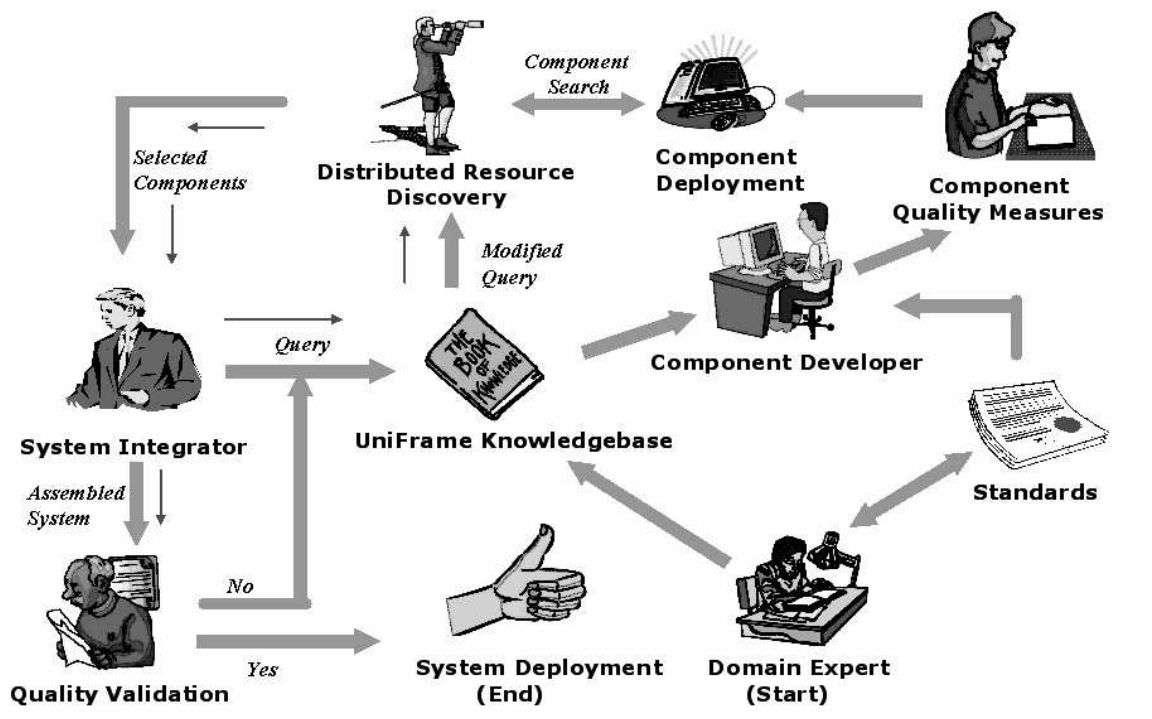


Figure 2.1 The UniFrame process

Therefore the component development and deployment phase begins with the "*domain knowledge base*" (refer to Figure 2.1) and the "*standards*" which are created and put in place by the "*Domain Expert*". The knowledge base includes the natural language-like specifications for a component for that domain – these consist of the computational,

cooperative and the auxiliary aspects and the QoS metrics of the component. The natural language-like specifications are then refined using the theory of Two-Level Grammar (TLG) natural language specifications [BAR00, VAN65] into a formal XML-based UMM specifications for the component. The derivation process also results in the generation of interfaces that incorporates all the UniFrame aspects of the component (discussed in Section 2.1.1). The "*Component Developers*" develop components in a component model of their choice and provide for the implementations for the computational and behavioral methods specified as part of the UniFrame knowledgebase. In addition the developer also refers to the UniFrame's QoS catalog [BRA01] in order to incorporate QoS metrics pertinent to the components of the domain the developer is developing components for. The "*component quality measures*" by the developer then include the empirical validation of the QoS of the component which determines the values for the identified QoS metrics of the component. The developer has to follow an iterative approach by refining either the UMM specifications or the implementation of the component. Upon a successful determination and implementation of the QoS of the component with respect to the QoS catalog, the developer now deploys his component on the network (refer to the "*component deployment*" in Figure 2.1). The component is now ready to be discovered and utilized in a larger application. Hence, the discovery of components by the infrastructure of the UniFrame could result in the components belonging to a diverse component models such as CORBA, J2EE, .NET, etc.

Automated System Generation and QoS-based Evaluation: The process of automated system generation and the evaluation of its QoS proceeds in an iterative manner. The process is outlined in the following steps:

a) The "*System Integrator*" who wishes to develop a DCS presents to the UniFrame system the query for the system with the required system characteristics. The system's query is then processed using the UniFrame domain knowledge base for the domain to which the system query belongs to. The knowledge-base consists of the requirements specifications and the matching design specifications. The latter specifies the type of components required to construct the queried system and the

interdependence between these components. The Composition/Decomposition model [SUN03] is then used to deduce the QoS of the required components. The set of functional and QoS-based search parameters are now available as a set of component queries (refer to "*Modified Query*" in Figure 2.1) to be processed by the URDS.

b) The "*Distributed Resource Discovery*" now performs a search within the scope of the domain of the component queries given to it. The search extends to all the heterogeneous component models and search is based on both functional and non-functional requirements specified as part of the query parameters. The set of discovered components is then returned to the system integrator.

c) Components that meet the query requirements to the maximum extent, are now selected to compose the system by the System Generator. The composition is carried out on the basis of the generation rules embedded in the system design specification outlined in the GDM. The components with the appropriate adapters (in case of heterogeneous components) form a software implementation of the targeted system. The adapter components act as the glue-wrapper bridging the gaps between off-shelf components chosen to implement the distributed system.

d) However, the system composed may or may not meet the system's requirements completely (refer to "*Quality Validation*" in Figure 2.1). This is verified using the event traces and a set of test cases. In case the system fails to match the specific constraints, the system generated is discarded. The process then may request additional components or attempt to refine the system query by adding more information about the desired solution from the problem domain. The process continues in an iterative manner unless and until the DCS is built satisfying the functional and QoS specifications of the system's query or the system integrator is satisfied with the generated system's test results. The system once has passed the quality validation tests it is ready to be deployed.

From the above discussion it can be inferred that the research issues incorporated within the UniFrame concept span across three main areas:

- Architecture-based Interoperability
- Validation of Quality Requirements
- Distributed Resource Discovery

The study of the UniFrame approach in the context of the .NET component model requires addressing some of the above issues and hence requires a deep entailed study combined with experimentation in each of the above fields. For the purpose of the study, the two main issues identified are Architecture-based interoperability and Distributed Resource Discovery. Both the issues involve the participation of components from different component models and hence were chosen as the concentration of the thesis.

The next section introduces the .NET platform followed by related work in the field of achieving interoperability between .NET and J2EE component model. This is followed by work in the field of "Distributed Resource Discovery" in the context of .NET as part of chapter 5.

2.2 Introduction to the .NET Platform

The following section describes the .NET platform by dividing it into two sub-sections. Section 2.2.1 provides the details of the basic .NET framework and its constituent parts. Since the focus of the study is primarily the distributed computing paradigm of .NET, Section 2.2.2 includes some details on its distributed computing paradigm, namely Web Services and Remoting.

2.2.1 .NET Framework

An application framework is defined to be a set of guidelines and specifications that provide platforms, tools, and programming environments for addressing the design, integration, performance, security and reliability of distributed and multi-tiered applications [WEB02]. Application development tools and application servers are built on top of these application frameworks. Microsoft .NET Framework [MIC02] is one such

application framework for client-server and Web-based applications. The framework is intended to make the development and consumption of Web Services a central part of distributed application development using .NET. The core components of the .NET Framework consists of:

Common Language Runtime (CLR): It is the runtime execution environment for the .NET applications. It provides services such as memory allocation, thread management, conversion of MSIL (Microsoft Intermediate Language) to the native platform code as well as enforcing security policies.

Framework Class Library: It provides a set of classes logically grouped into hierarchical namespaces that provide access to the underlying features of the operating system. It is aimed at providing a common set of APIs across all programming languages and thus enabling cross-language inheritance, error handling and debugging. This kind of interoperability is referred to as "Cross-Language Interoperability" in .NET terms. As long as the language targets the CLR, it can be integrated into the .NET application. Figure 2.2 depicts this fact.

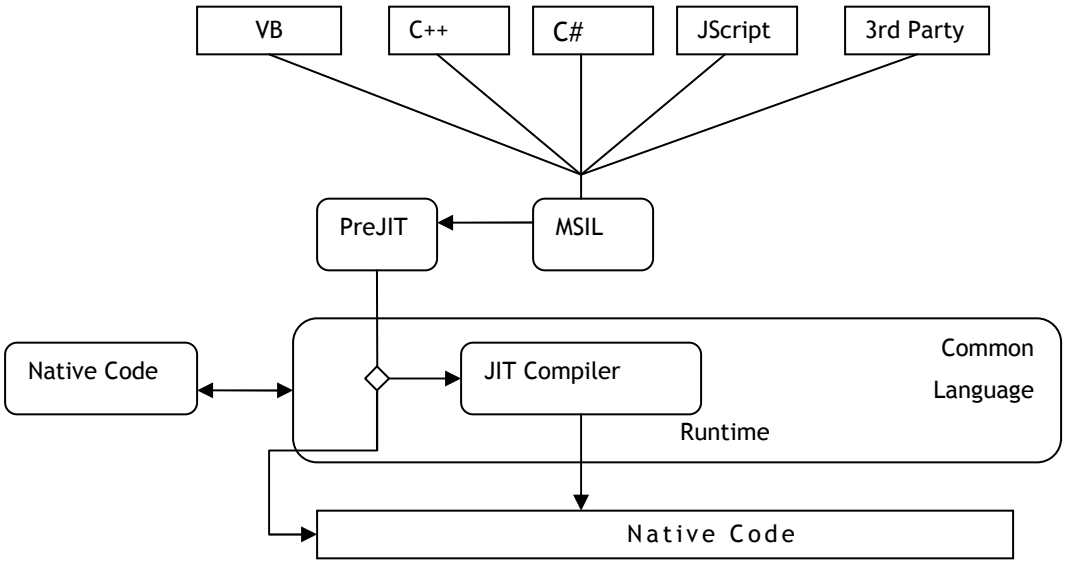


Figure 2.2 Cross language interoperability in .NET

ADO.NET: An extension to the ActiveX Data Objects, this data access technology also targets the Web.

ASP.NET: ASP.NET is the next version of Microsoft's ASP for building Web Applications, i.e. it provides a web application platform with services necessary to build and deploy web applications. Since it is a part of the .Net framework, it also incorporates the CLR and the Framework Class Library which enable these services. It enables two features for distributed applications: Web Forms and XML Web Services. Web Forms is a technology to build form-based Web pages. XML Web services enable the exchange of data using standards such as XML messaging and HTTP across firewalls in client/server and server/server scenarios. XML Web Services will be discussed in detail in the coming chapters.

The Common Language Specification: It is a set of rules that provides a contract governing the interoperability between language compilers and libraries [MIC03]. This is what enables multiple languages to run on the .NET framework thereby achieving the necessary cross-language interoperability.

Win Forms: Windows Forms is the .NET platform for Windows Application development based on object-oriented set of classes. Additionally, Windows Forms can act as the local user interface in a multi-tier distributed solution. Within a Windows Forms project, the form is the primary vehicle for user interaction.

Visual Studio .NET: It is the Microsoft tool that enables designing, building, testing and deploying of .NET applications including Web Services.

2.2.2 Distributed Computing in .NET

There are basically two computing models supported by .NET for cross-process communication. These are .NET XML Web Services and .NET Remoting [MIC01a].

2.2.2.1 .NET Web Services

An XML Web Service is a software component or an application that exposes its functionality programmatically over the Internet or intranet using the standard Internet protocols like Simple Object Access Protocol (SOAP) for inter-program communication and XML for data representation. The framework is based on the concept of a "service-oriented" architecture in which software is hosted as service. The underlying idea is: distributed applications are traditionally built using componentized software methodologies such as CORBA and DCOM. However this leads to a potential problem in the integration of distributed components. Since, each vendor provides its own set of interface protocols, use of any one of the technologies implies a homogeneous adoption of the technology which lacks a practical approach and universal adoption in today's disparate world. Web Services is an attempt to provide for a single unified infrastructure to integrate heterogeneous components, and one that scales to the Internet. Technologically, this is achieved by using message-based asynchronous technology and Web protocols such as HTTP and XML. Since it based on Internet standards, Web Services are loosely coupled. XML is the fundamental technology behind Web Services framework, which consists of the following main parts:

Wire format for Inter Process Communication: Simple Object Access Protocol (SOAP) is the Web Services' standard mechanism for specifying the format of data interchanged between the inter-communicating services. It represents one common set of rules about data and commands are represented and extended.

Description of Web Services: Web Services Description Language (WSDL) is an XML grammar is used to describe the capabilities of a Web Service since it deterministically specifies the set of data and commands that a service accepts.

Discovery of Web Services: DISCO – discovery protocol is Web Services' set of rules to define the protocol for developers to locate services' description documents.

Central Registration of Web Services: Universal Description, Discovery and Integration (UDDI) specification defines a framework for centralized registries which house the information necessary to locate, store and exchange information about WS. UDDI Business Registries store locations of the WSDL documents of Web Service in a phone directory like structure. Service consumers can use these registries to locate technical and general information about the service providers and then initiate transactions or collaborations with them.

Hence, the WS framework provides a kind of wrapper for applications, which allows them to interoperate on the Internet. This wrapper provides a standardized means of describing WS, and what it does; publishing it to a registry, so that it can easily be located and exposing an interface, so that the service can be invoked – all in a machine-readable format. Hence interoperability is achieved between all clients and servers who understand XML.

ASP.NET is Microsoft's IIS-hosted infrastructure that supports industry standards for WS. [MIC05]. It accomplishes this by providing a programming model based on mapping SOAP message exchanges to individual method invocations.

2.2.2.2 .NET Remoting

The .NET Remoting framework is another approach in the .NET paradigm which allows the development of distributed applications and could includes web services as well. In general, it is the process of programs or components interacting across certain boundaries. Those contexts could be different processes or machines. Remote objects provide the ability to execute methods on remote server, passing parameters and receiving return values. The remote object always stays at the server, and only a reference to it passed around among other machines.

Remoting implementations generally distinguish between “remote objects” and “mobile objects”. The former provides the ability to execute methods on remote servers, passing parameters and receiving return values. The remote object always resides on the server, and only a reference of it is passed to the other machines.

When mobile objects pass a context boundary, they are serialized (marshaled) into a general representation either a binary or human readable format like XML – and then de-serialized in the other context involved in the process. Server and the client both hold copies of the same object. Methods executed on those copies of the object will always be carried out in the local context, and no message will travel back to the machine from which the object originated. In fact, after serialization and de-serialization, the copied objects are indistinguishable from the regular local objects, and there is no distinction between a server object and a client object [MIC04].

Remoting Architecture: Remote objects are accessed through channels. Channels are transport protocols for passing the messages between remote objects. A channel is an object that makes the communication between a client and a remote object across application Domain boundaries. The concept of an “application domain” in the context of .NET is particularly significant in understanding the distributed architecture of the .NET based discovery service. The .NET’s terminology of an application Domain will be further discussed in detail in Chapter 5, Section 5.3.1. The .NET framework implements two default channel classes:

- HTTP channel: Implements a channel that uses the HTTP protocol.
- TCP channel: Implements a channel that uses the TCP protocol.

A channel takes a stream of data and creates a package for a transport protocol and sends to the other machine. A simplified architecture of .NET Remoting is shown in Figure 2.3. The figure depicts that a remote object is hosted within the context of the server’s application domain (to be discussed later in Section 5.3.1) via the .NET Remoting System (or the Remoting infrastructure). The remote object lives and functions within the boundaries of its application domain.

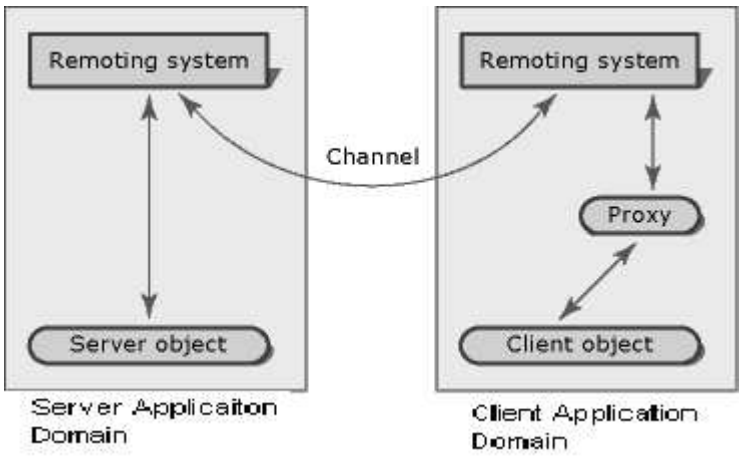


Figure 2.3 .NET Remoting architecture [REM04]

On the other end, the client creates a new instance of the server class knowing the URL and the type of the remote object. The Remoting system creates a proxy object that represents the class and returns to the client an object a reference to the proxy. When a client calls a method, the Remoting infrastructure handles the call, checks the type information, and sends the call over the channel to the server process. A listening channel picks up the request and forwards it to the server Remoting system, which locates (in case of server-activated objects or creates in case of client-activated objects, if necessary) and calls the requested object. The process is then reversed, as the server Remoting system bundles the response into a message that the server channel sends to the client channel. Finally, the client Remoting system returns the result of the call to the client object through the proxy. The process is similar to other remote communication paradigms currently existing such as stub-skeleton mechanism. Remoting offers certain advantages over the Web Services.

- Multiple protocol support including high-speed binary over TCP - faster than SOAP over HTTP.
- Support for activation and lifetime control of remote objects by the client (similar to DCOM).
- Support for passing objects by reference and by value.
- Support for callbacks.

- Support for additional context information specific to .NET.
- Support for events.
- Provision for one-to-one mapping between the class and type hierarchy. Web services and SOAP do not support such an object-oriented mechanism for accessing remote objects.

The other details of the .NET Remoting paradigm become clearer as the study progresses.

2.3 Resource Discovery

As indicated in Section 2.1, one of the main challenges of the UniFrame's objective is the issue of distributed resource discovery. The discovery of resources plays a significant role in locating, accessing, retrieving and managing pertinent resources from distributed and heterogeneous networks. Such a facility is extremely important for integration platforms such as UniFrame in enabling the automation of the process of assembling distributed system out of heterogeneous components. Such a framework entails the need for an infrastructure that can dynamically discover the presence of new components in the search space which utilize and offer services, and allow for the selection of components meeting the necessary functional as well as non-functional requirements (such as desired QoS). The infrastructure also needs to provide translation capabilities for specific models [SIR01]. The URDS is a proposed architecture with such capabilities. There are other existing resource discovery services which seem to provide similar functionalities. Based on the basic underlying concept the discovery services can be categorized into two main groups, a) Lookup Services or Directory Services, and b) Discovery Services.

Lookup (or Directory) Services: Lookup services imply passive services in which the service requestors initiate a request to obtain the information of a required service. Such services require the presence of some kind of a directory (or agent) to process the incoming requests. UDDI Registry, CORBA trader services [OMG00, OMG01b], LDAP [WAH97], Domain Name System [MOC87], etc. fall under this category.

Discovery Services: The services under this category adopt a more active nature and less initiative on behalf of the service seekers. The services allow components to discover each other in a spontaneous manner based on service descriptions with little or no human administrative intervention. Service Location Protocol (SLP) [GUT99a], JINI [SUN01a], Ninja project using Secure Service Discovery Service (SSDS) [CZE99, NIN02], etc. protocols employed by services belong to this category.

Internet Resource Discovery Protocols also define services to grant users access to distribute information retrieval systems encompassing data for millions of web sites, networks and users. Some of these are, Wide Area Information Servers Project (WAIS), Archie and Gopher. Since the service acts a lookup based service, it alone cannot meet the needs of dynamic discovery of components and other aforementioned needs of integrating platforms like UniFrame. One of the other promising approaches for discovery process is the Monitoring and Discovery Service (MDS) that forms the basis of the Grid discovery service and is mainly employed for computational resources deployed on the Grid. It also supports searching for resources by characteristics. However, unlike URDS, the performance of a query to the MDS cannot be predicted with a pre-defined formula [GLO04] and is depended on the complexity of the associated hierarchy of its components, Grid Resource Information Service (GRIS) and Grid Index Information Service (GIIS).

Both the categories of services pose the important question of heterogeneity. Not all these services address the issue in comprehensive detail. For example, the CORBA Trader services offer directory services only for CORBA objects and the JINI discovery services spontaneously discover only other JINI enabled devices. UDDI serves as a look-up service only for components which have been leveraged to the Web Services framework. For these services to be logically compatible, they need to be mapped by implementing equivalent protocols which defies the underlying goal of a “universal” discovery service. The URDS architecture proposes to tackle this issue of interoperability by providing discovery and directory services for components developed using different

component models [SIR01]. The URDS combines the notion of discovery and directory services by means of achieving a federation of its services. The ICB acts as the directory for brokering the clients' requests and responds with a list of matching services. These services can belong to diverse component models. This is achieved by means of the ICB, Headhunters and Active Registries (to be discussed in Chapter 5). In addition, an ICB can also communicate with another ICB for handling the clients' queries and increasing the search space. This ICB can be homogeneous or heterogeneous. The communication is enabled by means of a "Link Manager" (to be discussed in Chapter 6) which, with the help of the Glue-Wrapper generator framework of the UniFrame can interoperate across heterogeneous discovery services. Hence, the URDS addresses the issue of heterogeneity with respect to the discovered components as well as other discovery service instances.

The third area of study explored in terms of the .NET component model placed in the context of UniFrame is addressing the problem of heterogeneity within the UniFrame paradigm and applied and experimented with respect to .NET. The issue is discussed in further detail in Chapter 3 and 6. Since the experimentation undertaken to tackle this issue has been with two component models, namely .NET and Java RMI (whose prototypical setup existed at the time of the study making it an obvious choice to experiment and deal with), the next section discusses some of the commercial bridges that aid in the .NET and Java interoperability. Some of the other mechanisms to achieve this interoperability will be further discussed in chapter 3 while addressing the issues of "*Problem of heterogeneity*".

2.4 Commercial Bridges for .NET-Java Interoperability

Both .NET and Java (RMI or any other Java platform such as Java 2 Enterprise Edition) are component models that exist widely in the field of application development. While Java has been prevalent for a long period of time, .NET as rapidly emerged as Microsoft's component model inherently supporting the notion of Web Services. Co-existence of both these models necessitates the means by which efficient integration can

be achieved in a heterogeneous computing environment. This has led to the development of a number of commercial software products that enable .NET and Java components to interoperate. These products that achieve such functionality are termed as “bridges”. Examples of such bridges (specifically for.NET and Java) are JNBridge [BRI01], iHUB bridge [BRI02], Ja.NET bridge [BRI04].

However, it is important to understand as to the factors that need to be considered while incorporating any bridge, since every bridging software has an associated set of incorporation requirements that play a significant role in deciding a bridge that meets an application’s integration requirements in the most appropriate manner.

Section 2.4.1 outlines the basic principle behind the bridging products experimented with during the study. It should be noted that the study is comprehensive enough in selecting the most prominent bridging solutions (for .NET and Java) that do not limit the bridge’s functionality to solely Web Services as the interoperability mechanism. Cape Clear solution [CAP04] falls into the category of such bridges, but is not included as part of the study. The following sections discuss some of the bridges that were experimented with during the course of the study and the experience gained as a result. The use of bridges is leveraged in Chapter 6 to provide interoperability between heterogeneous discovery services under the UniFrame framework.

2.4.1 Underlying Principles of Bridges

The study of bridges reveals a fact that the almost all the bridging products do not have one set of compilation cycles and proxies that could connect a .NET and a Java component in a bi-directional way. This implies that the steps involved in making a .NET component invoke the functionality of a Java Server are different than those involved in a Java component invoking a .NET server. Hence, the bridging softwares internally incorporate two one-way bridges – either .NET to Java or Java to .NET. Thus the term “bridging software” instead of “bridge” would be used by the next few sections.

Every bridging software consists of a *runtime environment* or a kind of a service which requires to be running on one of the platforms – either the client or the server. In some bridges this can vary and might require background services on both the platforms.

At the core of every bridging technology lies a *proxy generator*, which generates the necessary proxies to enable the heterogeneous remote objects to communicate. While some bridging softwares consists of a single generator to do so, in other bridges, there is a separate generator to generate each of the .NET proxies (of Java objects) and Java proxies (for .NET objects). It is the capability of the generator that determines the nature of the proxies generated and hence determines the extent of interoperability that the bridge can provide. For example, if the generator is unable to generate mappings for user-defined classes in one platform, the proxies will be devoid of these mappings and will not allow the object in the other platform to be able to identify them limiting the scope of the communication between the heterogeneous objects.

Even though the representation of the generators and the runtime environment is different for each bridge, the underlying principle remains the same. Figure 2.4 depicts the general scenario employed by the bridges with the example of a .NET client and a Java Server.

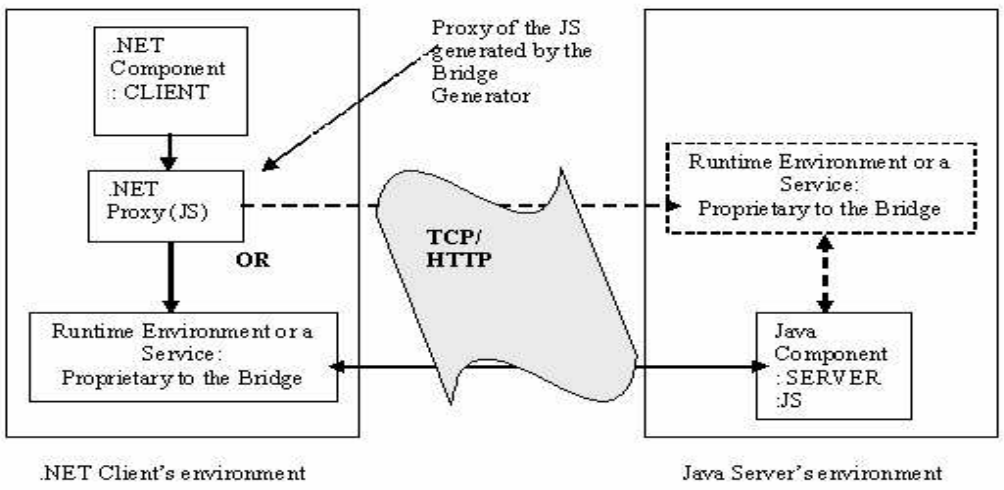


Figure 2.4 General scenario employed by .NET-Java bridges

The figure shows that the runtime environment proprietary to a bridging software needs to be present either on the client's machine or on the server's platform (or both in some cases). The generator of the bridging software generates the server's (JS in the Figure 2.4) proxy which is then used by the client (.NET client in this case) as a handle to the remote server. The client needs to be re-compiled so as to access the necessary proxies. The generated proxies in turn utilize the runtime environment of the bridge to forward the calls of the client to the remote server. The runtime environment acts as the communication engine between the two component models by providing the necessary type mapping and the APIs for the proxies to access. The client's call then passes to the remote heterogeneous server via means of the bridge established.

The next two sections illustrate the concept of bridges with the help of two such bridging technologies – iHUB and Ja.NET. Both the bridges have been experimented with during the study and Ja.NET has also been incorporated to achieve the interoperability between heterogeneous discovery services (as will be explained in Chapter 6).

2.4.2 iHUB Bridge

This .NET and Java Integration product constitutes a number of individual bridges that are installed independent of one another. These are, Java to .NET (J2N) Bridge, .NET to Java (N2J) Bridge, Java to Windows (Java2COM) Bridge, Windows to Java, (COM2Java) Bridge, and XML/Web Services Bridge. And as mentioned earlier, each way communication, Java to .NET (J2N) and .NET to Java (N2J) constitutes a separate bridge. In accordance with the focus of the section, only the first two bridges would be discussed here, namely the J2N Bridge and N2J Bridge. Each of the two bridges, are made up of a different set of modules. The common module is however the "proxy generator" but requires a different instance for both J2N and N2J Bridge; a fact that the bridge can be only one-way.

2.4.2.1 J2N Bridge

iHUB's J2N bridge enables Java applications to create and invoke methods on .NET Remoting objects. It comprises of a Proxy Generator and the Client APIs (.NET Remoting equivalent APIs for Java) and the Channels and the Communication engine). All these modules are combined into a ihub.jar file which is essential for iHUB to function properly. This file is the equivalent of the runtime environment that was mentioned in Section 2.3.1. The following figure depicts the modules involved in a Java component invoking a .NET Remote object.

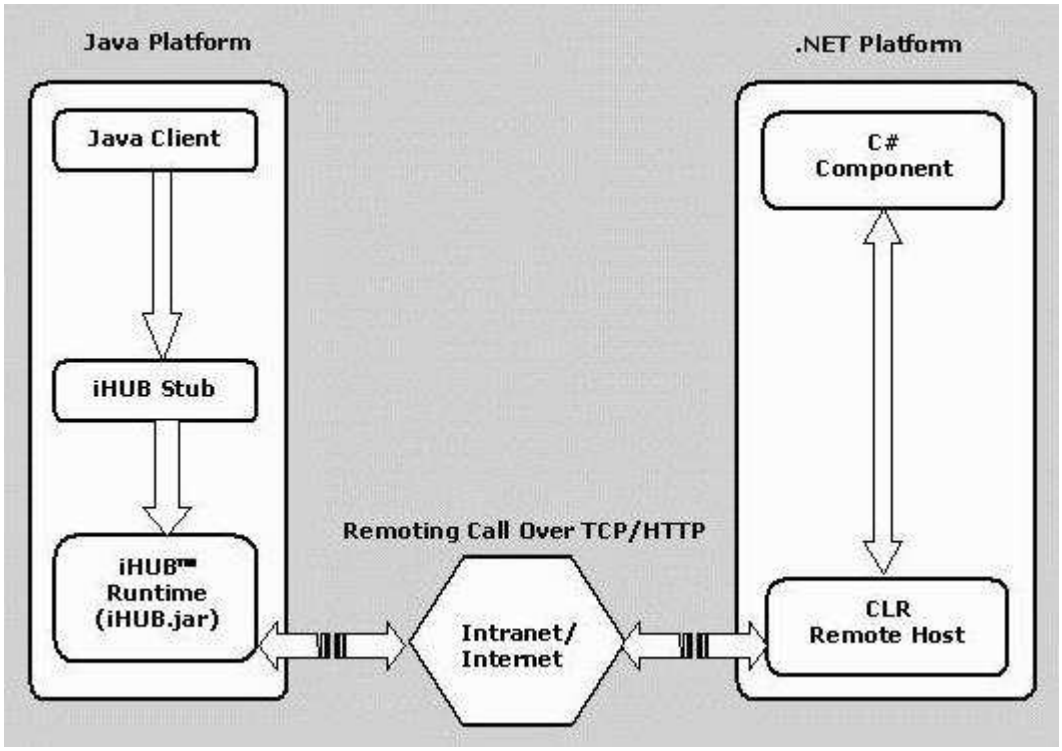


Figure 2.5 Java and .NET interoperability using iHUB's J2N bridge [BRI03]

As can be seen from the figure the basic concept is similar to what was depicted in Figure 2.4. The figure shows a .NET component (implemented using the C# language) exposed as a remote object on a remote host. The Java client can access the remote object through the iHUB stub for the remote object. The Java client makes use of the iHUB

Runtime (ihub.jar) which is being called by the iHUB stub. It communicates over the network to the CLR remote host & consequently with the .NET component.

J2N Proxy Generator: iHUB's Proxy Generator is used to generate local Java proxies (.java or .jar files) for remote .NET components. The Java application makes all the method and service calls on the local proxy. The proxy in turn talks with the iHUB Communication Engine which facilitates communication with the .NET object using the .NET Remoting packet format. The communication details and the protocol complexities are handled by the iHUB.jar file. Not indicated by the Figure 2.5, but the proxy generation is carried out via a process in which iHUB service must be installed on the .NET platform on which the remote .NET object whose proxies are to be generated resides. The J2N proxy generator invokes the iHUB Service to get the information about the .NET object. The Java proxies/stub is generated as a result, contained in a jar file. The Proxy generator can be invoked from a remote Java platform as well, in which case the proxy file is transported over the wire. The other modules of the J2N Bridge constitute the Remoting API for java, Channels and the Communication Engine.

Remoting API for Java: The J2N Bridge provides a set of client APIs to the Java applications in order to access the Remoting objects. These interfaces are identical to the Remoting API's that are available in .NET.

Channels: As indicated in Section 2.2, the Remoting model incorporates the concept of Channels which act as the means of message transportation between .NET Remoting objects. iHUB bridge makes this concept available by the its Remoting APIs to the Java client as well.

Communication Engine: It facilitates the communication between the two frameworks by providing multi-client access to the .NET server, wrapping the Java calls into the Remoting packet format, providing the type mapping between Java and .NET Remoting including the exception and error handling types.

2.4.2.2 N2JBridge

The N2J bridge of iHUB supports the .NET clients to invoke the functionality of the Java server. The Java objects supported by the bridge include Java classes, JavaBeans, Servlets and EJBs. The Figure 2.6 depicts the process of communication between .NET and Java object through the N2J Bridge.

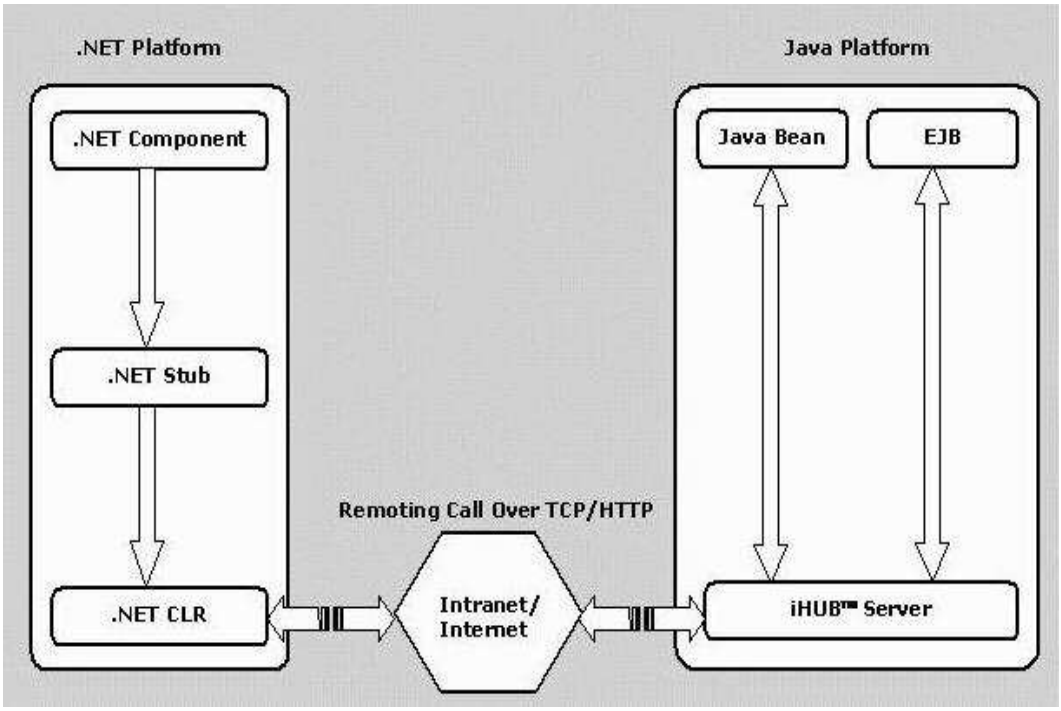


Figure 2.6 .NET and J2EE interoperability using iHUB's N2J bridge [BRI03]

The one-way N2J bridge again indicates the same underlying principle as discussed in Section 2.3. The figure can be explained with the help of the following three main parts which constitute the N2J bridge.

N2J Proxy generator: The Proxy Generator generates the .NET proxies (for the remote Java object) and compiles them into .NET assemblies to be accessed by the .NET client. These assemblies can then be invoked by the .NET using the Remoting API of the .NET class library

iHUB Server: This server is analogous to the runtime environment provided by a bridge. In this case it resides on the Java server. All the methods invoked by the .NET component gets routed through the iHUB Server, which hence, serves as the .NET CLR on the remote machine. It provides all functions of the .NET CLR for Remoting objects including object management, lifetime control, method invocation, etc. The server also supports both TCP and HTTP channels, and thus, both binary and SOAP formats. Hence, the access to the remote Java object is transparent to the .NET client.

Registration of Remote Java Objects: Since the Remoting calls of the .NET client are processed by the iHUB server, it is important for the Java objects on the server's platform to be registered with the iHUB Server. This mechanism creates a one-to-one mapping between the Java object and the properties provided by a corresponding .NET Remoting class. The properties range includes lifetime, mode of communication, channels and lifetime leasing of Java objects.

2.4.2.3 Observations

Some of the following facts that can be concluded from the above discussion are that, a) whether it is the J2N bridge or the N2J bridge, both the bridges require a certain level of alterations and compilation cycles on the client's end, b) there are services and runtimes which are required to be present on the client's or the server's side, and c) in both the J2N and N2J bridge, the communication between the components is carried out by the use of the Remoting paradigm (whether the use is of binary or the SOAP protocol). And based on whether the client belongs to the Java or the .NET component model, iHUB runtime is required on that particular host. For example, if the client belongs to the Java component model (Figure 2.5), the iHUB runtime needs to be present on the client's host to convert the calls made by the Java object in a corresponding Remoting call to the .NET Remoting server. And reverse is the case for a .NET client (Figure 2.6). The iHUB server is hosted on the Java server's end in order to process the

incoming Remoting calls. Hence, the iHUB bridge leverages the .NET Remoting paradigm at the basic level to achieve interoperability.

The experimentation of the iHUB bridge was carried out with the Java RMI component model and was utilized to achieve interoperability between Java RMI headhunters and .NET Active Registry. In addition to the above mentioned observations, the other facts that have become evident after carrying out the experiments are: a) the bridge is limited in its ability to serialize objects' interfaces (to be discussed in detail in Chapter 5), b) in order to pass data by encapsulating it into built-in objects of a component model, such as ArrayLists, Hashtables etc, the bridge requires a lot of customization code on the part of the client application. This requires additional efforts from the application developer in order to incorporate the bridge's functionality for achieving the necessary integration and c) the iHUB APIs cannot be used by the Java client if the client is developed on the RMI component model. The client only utilizes the proxy generated by the proxy generator and needs to write custom code in order to access the proxy. This adds another level of complexity to the use of bridge and could be deduced only after experimentation with the bridge.

2.4.3 Ja.NET Bridge

The Ja.NET bridging software is provided by the Intrinsyc Inc. and like the iHUB bridge, it also leverages the .NET Remoting platform to integrate to integrate Java-based application and .NET components. In accordance with the support for elements of the .NET Remoting framework, the bridge supports HTTP and TCP/IP protocols and either SOAP or binary data formatting. It supports the Java and .NET component models for various different servers such as EJBs, Java Servlets, .NET applications within the IIS server etc. The bridging product is made up different entities which can be grouped into the following categories according to the discussion of the underlying principles in Section 2.4.1.

Bridge Runtime environment: This category consists of the runtime environment of the bridge which is needed for its operation. The "Ja.NET Runtime" falls into this category. The software also consists of a tool (called *Janetor*) required to bootstrap the Ja.NET Runtime and provides it with details such as, a) the location, type and channel format of remote objects, b) hostname, port, assembly name and type of local objects, c)licensing information, etc. Hence, it prepares the runtime to process the incoming calls from a .NET client or outgoing calls from a Java client. The scenario is similar to the iHUB bridge where the iHUB's runtime environment is always required the on the Java component's end to be able to process the incoming/outgoing calls. Figure 2.7 and 2.8 depict the .NET and Java interoperability using Ja.NET bridge in case of Java Client-.NET Server and .NET Client-Java Server situations respectively.

Figure 2.7 depicts a Java Client invoking the Ja.NET Runtime (by means of proxies which form a part of the Ja.NET runtime in the figure) to obtain a reference to the Client Activated Object (CAO) hosted on the .NET platform. The Ja.NET Runtime is configured with the contact details of the remote object and hence forwards the call of the client to the remote server (of the name "Factory") of the type Singleton that is hosted by the server side and returns a reference to the CAO object.

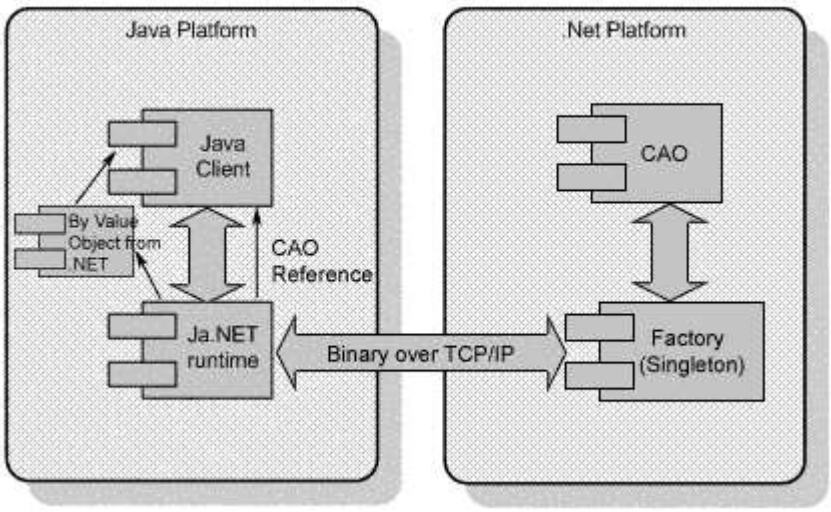


Figure 2.7 Java client - .NET server interoperability using Ja.NET bridge[BRI04]

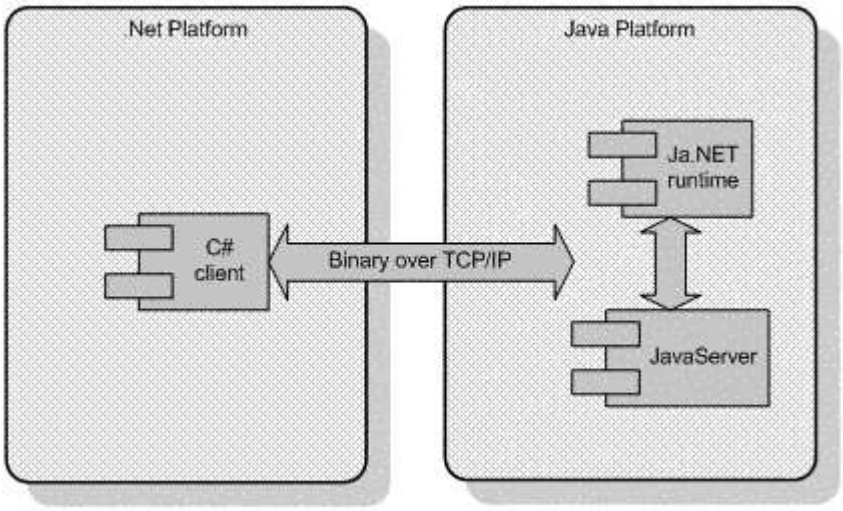


Figure 2.8 .NET client – Java server interoperability using Ja.NET bridge [BRI04]

Both the figures above show that since it is the .NET Remoting paradigm that is leveraged by the bridge, the Ja.NET Runtime is executed on the platform where the Java component resides. Figure 2.7 is in correspondence with Figure 2.5 and Figure 2.8 with Figure 2.6 in this respect (Ja.NET Runtime mapped to the iHUB Runtime or server). Hence, both the bridges operate on the same underlying principle with different implementations and capabilities. The Proxy generator (explained next) creates the necessary proxies of the server object for the client to access.

Proxy Generator: The Proxy Generator for the Ja.NET consists of three entities: a) GenService, b) GenJava and c) GenNet. The GenService is a Windows Service component and is only deployable on the Windows platform. GenJava and GenNet are two tools which aid in the generation of the proxies with the help of the GenService. GenJava generates the Java proxies of a .NET server to be accessed by a Java client and GenNet generates the .NET proxies of a Java server for a .NET client. The GenJava tool uses the service to read the .NET assemblies and the GenNet tool to write .NET assemblies. Thus, the functionality of the GenService depends on which of the tools is invoking it and is now outlined in the following paragraphs.

GenNet Tool and GenService: GenNet uses GenService to create a .NET assembly from the input Java classes. A summary of the steps taken by GenNet when it is invoked to generate a .NET assembly is as follows:

- GenNet analyses the Java component for its incorporated types.
- XML is used as the description language for the analyzed Java types by the GenNet, which then sends the description to the GenService.
- GenService generates the necessary .NET assembly based on the XML description at a location relative to the location of GenService.

GenJava and GenService: GenJava uses GenService to obtain metadata about a .NET assembly. The Java proxies are then generated by GenJava and output to a location relative to where GenJava resides. A summary of the steps taken by GenJava when it is invoked to generate Java proxies is as follows:

- GenJava sends a notification event to the GenService providing the location of the .NET assembly for which the proxies need to be generated.
- GenService reads the assembly metadata and sends back an XML description.
- GenJava generates Java proxy based on the XML description.

Hence, the actual proxy generation in this case is done by the GenJava rather than the GenService.

2.4.3.1 Observations

Most of the observations for this bridge are the same as those for the iHUB bridge such as, a) the underlying principle is the same as indicated, b) Ja.NET also leverages the .NET Remoting platform for achieving interoperability, c) the software is also made up of two one-way bridges and requires a certain level of alterations and recompilation cycles by the client invoking the remote method, d) there are services and runtimes which are required to be present on the client's or the server's hosting machine. The Ja.NET runtime is required on the Java component's end and since the GenService is a Windows

service it places a constraint on the component developer to utilize its functionality within a certain restricted environment.

The experimentation with the Ja.NET bridge also revealed that the bridge is limited in its functionality to serialize objects' interfaces. However, the bridge is more efficient in terms of convenience and ease it provides in serializing data encapsulated in ArrayLists, Lists and Hashtables etc. The runtime provides direct mapping between these types of the respective component models with few or no additional steps required during development of the code. It is due to this reason that the bridge was chosen as the preferred method of interoperation for the experimentation of Chapter 6 and the third goal of the thesis.

This chapter provides the previous work which lays the foundation of the thesis - UniFrame and .NET component model – the two areas which the study targets. Also, related work in the field of discovery services and commercial bridges which the thesis later incorporates has been discussed. One important aspect of studying a component model within an integration platform is to address the issues that can subsist with the co-existence of other component models. Thus, it becomes important to address the problem of heterogeneity in this respect. The next chapter introduces this problem in the context of UniFrame and entails details about the approach that would be incorporated in the later chapters to tackle interoperability.

3 PROBLEM OF HETEROGENEITY

Commercial bridges are discussed as an interoperability mechanism in the previous chapter. Also the focus was to analyze this interoperability mechanism in the context of .NET and Java RMI keeping in line with the concentration of the study and the ease of experimental validation with respect to the existing prototypical implementation in Java RMI. The aim was to provide a background for the manner in which the problem of heterogeneity tackled as part of the study. However, since the goal of the thesis does not confine itself to only providing a .NET-centric solution, it becomes important to discuss this problem in general in the domain of the UniFrame approach. The assumption of the UniFrame approach is that the components will be developed and deployed independently in a networked environment to be discovered and consumed by interested service requestors. The autonomous and independent nature of the process implies that the components can be heterogeneous and an integration platform requires addressing these differences in order for a successful system composition. However, the term "heterogeneity" comprises of different types of existence. These are: a) Syntax/Signature heterogeneity (differences in the components' interface signature), b) Semantic heterogeneity (differences in the meanings of the provided/required interfaces of the components), c) Protocol differences (ordering and blocking constraints that determine the availability of services provided by components) and d) differences in the QoS specifications of components. Heterogeneity also exists due to the differences in the technological component models used to develop these components. This kind of heterogeneity can encompass one or more of the aforementioned heterogeneities as well. For example, components developed using different component models can also be at the same time have different interfaces (both in their signature and semantics) and different

protocols of communication. One of the sub-goals of this thesis attempts to address this problem in the context of UniFrame and this chapter provides the necessary outline for it. Section 3.1 identifies the different points of incongruity for this kind of heterogeneity and those which should be tackled by a chosen interoperability mechanism. Section 3.2 then lays out the different areas as to where the heterogeneity can exist within the UniFrame paradigm followed by Section 3.3 which analyzes the various mechanisms for handling this heterogeneity (particularly for .NET and Java RMI component models). However, any approach which the UniFrame incorporates should be extensible to incorporate different interoperability mechanisms and component models. A result of the investigation undertaken for addressing this problem, as part of the study, is the suggestion of the concept of "connectors", which advocate a promising approach for the realization of the concept of glue-wrapper generation for automated system generation under the UniFrame approach. Section 3.4 discusses as to why.

3.1 Points of Identification

As discussed in Chapter 1 and Section 1.1, when components belonging to different COTS component models need to be integrated, there could arise various incompatibilities due to the differences in the underlying models, making it difficult if not impossible to handle the interactions between them. There can be the following three identified incompatibility points [RAP01].

1. Different Interface Approaches and Implementations

One of the basic elements of a component is its interface. Through its interfaces objects expose their functionality. An interface consists of a description of a group of possible operations that a client can ask from the object. A client always interacts with the interface of the object and never with the object itself. Interface allows an object to appear as a black-box. Different approaches and implementations of object interfaces make them invisible to clients of other technologies.

Both CORBA and DCOM use special Interface Definition Languages (IDLs), for the interface specification. Java RMI uses the Java Language to define the interfaces. Similarly, .NET Remoting uses any of the supported langue such as C#, VB etc to specify the interfaces. In DCOM, every interface has a Universally Unique Identifier (UUID), called the Interface Identifier (IID) and every object class has its own UUID, called the Class Identifier (CLSID). Moreover, every object must implement the IUnknown interface. When using the Java language to specify DCOM objects every Java class implements that interface behind the scenes through the Microsoft Java Virtual Machine (MSJVM). In Java RMI, the interface must be declared as public, must extend the java.rmi.Remote and each method must declare java.rmi.RemoteException in its throws clause. Similarly, every remote object using the .NET Remoting object model must implement the class MarshalByRefObject.

2. Different Object References and Storage

When a client wishes to interact with an object then it must first retrieve information on the object's interface. A client's underlying technology must recognize an object's name; it must know where to look, and how to retrieve its information. In other words, the client must know as to how the required object's technology stores and disseminates information. If the client's technology does not have that kind of ability then it is impossible for the necessary information of the required object to be found.

In CORBA, the IDL compiler generated the appropriate client stubs and server skeletons for a client to deposit a static invocation to the requested object. Moreover, all the required information is stored in the Interface Repository through which the client can get the run-time information for a dynamic invocation. The client is searching for the needed methods using the object's name reference and invokes it statically, through the client stub interface, or dynamically, through the dynamic invocation interface, depending on its runtime knowledge. For the interaction to be possible the CORBA server program must bind the server object using the CORBA Naming Service. Prior to

the above interaction, the CORBA server and the CORBA client program must first initialize the CORBA ORB through the ORB.init() method.

Using the Java RMI, in the server side program, one creates the server object and binds it to the RMI Registry using the Naming.Rebind() method by assigning a URL-based name. On the client side, the Java RMI client gets a reference from the server's registry using the URL-based object's name through the Naming.lookup() method.

.NET Remoting utilizes a registration of the built-in RemotingConfiguration class of the .NET Framework Class Library. And as explained earlier, the registration is confined to the application domain in which the host application hosts the registered object. Each object makes the following parameters known to the RemotingConfiguration Class:

- Unique URL of the object
- The Object Type

A client knowing the URL of the object can invoke the RemotingConfiguration class in order to obtain a reference to the remote object and then invoke its functionality.

3. Different Protocols

Another basic element in distributed object interactions is the protocol used for the data transmission. In this case, the term "protocol" not only denotes the transport-level protocol, i.e., TCP/IP but also includes the presentation and session level protocols supported by the Request Brokers (RBs). The transport-level protocol is responsible for transmission of the data to the end-point. The presentation and session level protocols are responsible for the formatting of the data transmitted between different RBs from a client to an object, and vice-versa. According to Geraghty et all [GER99]: "Although the client and the server may speak the same protocol, it is critical that they speak the same language, or higher-level protocol."

Identification of the kinds of incompatibilities that can exist while connecting components of different component models leads to another question – can heterogeneity exist in UniFrame only at the level of system generation? Are there any other areas where this issue needs to addressed in the UniFrame context? The following section attempts an answer to these.

3.2 Heterogeneity within UniFrame

Some of the areas that can be recognized as possible areas of existence of heterogeneity within UniFrame are as follows:

Within the principals of the URDS: As discussed in Chapter 2, Section 2.2, most of the discovery/registry services do not assume the presence of other models. The interoperability that they provide is limited mainly to the underlying hardware platform, operating system, and/or implementational languages. The URDS is an attempt in the direction of providing for a "service" discovery model that is dynamic and encompasses services developed in diverse distributed computing models. This requires the headhunters of the URDS to be universal in nature and hence should be able to interoperate across disparate networks, with heterogeneous entities, which may be other headhunters as well. Headhunters also communicate periodically with the Active Registries to collect the registered services' descriptions. The Active Registry of a particular component model extends the native registration mechanism of that model (Active Registry will be discussed in detail in Chapter 5). Thus, every component belonging to a specific component model would register with an Active Registry of its own component model. This facilitates the fact that the registration mechanism is not uniform since it builds upon the indigenous technology of the underlying component model. Hence, there would be as many different implementations of Active Registry as there would be the component models, whose discovery and integration UniFrame supports. Enabling communication of all such Active Registries with other principals of

the URDS, such as the headhunters, also requires handling interoperability between heterogeneous component models.

Linking multiple instances of URDS: In order to achieve scalability there arises a need to link different instances of URDS to form a federation of UniFrame discovery service. The federation allows the search space of a URDS instance to span across multiple ICBs and hence provide for a more scalable and comprehensive solution to dynamic discovery. Since URDS is a platform-independent architecture, the realization of different entities of a URDS instance need not conform to one particular component model. There could be a disparity between different URDS instances depending on the configuration of the service which can depend on different factors. This introduces a level of heterogeneity within and across URDS instances. Thus, federation of UniFrame discovery service also needs to tackle interoperability between heterogeneous URDS.

System Composition: After the components have been discovered by URDS, based on a certain criteria, they need to be integrated. It is possible that the chosen components belong to any of the component models. Integrating a system out of these heterogeneous components is a major challenge as it requires resolving the issue of heterogeneity. Not only the system composition should resolve the basic architecture-level heterogeneity but since the composed system must also meet the system's quality of service criteria, the glue-wrapper code that interacts between the heterogeneous components should also have an associated quality of service within the QoS constraint of the system being composed. This necessitates the need for a "gluing mechanism" which can achieve a complete amalgamation of QoS constraints and other system requirements, incorporating the known "interoperability mechanisms" to produce a system meeting the necessary query conditions.

Before proceeding to the concept of "connectors" which is the suggested approach for such a "gluing mechanism", the next section lays out the different options

for achieving interoperability between different component models. For the sake of compactness the mechanisms are discussed within the context of .NET and Java RMI.

3.3 Different Mechanisms for Interoperability between .NET and Java

There are a multiple of methods to achieve interoperability between Java and .NET. The following section illustrates a few of the methods to do so, namely, interoperability using Web services, binary communication, CORBA and resource tier solutions [INT01].

Web Services [WEB03]: Web Services were introduced as one of the distributed computing paradigms of .NET. The issue would be discussed in further detail in the next chapter. However, in this section, Web Services would be discussed solely from the view of an interoperability mechanism rather a whole distributed paradigm itself.

Interoperability using Web Services can go across firewalls and proxy servers. In addition, the SOAP and WSDL (introduced in Chapter 2 under related work) prove to be an extensible and flexible format and standard of data representation enabling further interoperability. However, there are few potential problems that can be posed with Web Services for J2EE and .NET Interoperability.

[INT01] Data passed using Web Services inherently depends on XML and XSD [SKO03]. Web Services pass SOAP messages encapsulating this XML data to another entity using HTTP. Serializing data from in-memory objects to XML can be relatively expensive, both computationally and for the resulting size of the data. Taking large data objects and converting them to ASCII based XML representation can result into large documents in certain scenarios. In case where the communication involves single message or document across the network each day, this might not be a perpetual problem. However, when thousands of messages are exchanged per second between two systems, the additional overhead of message processing can be a consideration.

The second issue with Web Services interoperability is the HTTP protocol itself. HTTP is a request-response protocol. The client makes a request and expects a response in a within the lifetime of the call. Some applications work better with an asynchronous model – for example, a loan or credit card application process may take several hours to complete. In such a case, the client holding the call channel open for that duration is inappropriate. There are other alternatives to this problem such as the use of asynchronous calls to poll the server regularly to see if the request operation was completed. However, polling have certain inefficiencies which require other transport protocols. These are TCP channel for the client to listen to the response from the server or use of SMTP messages by the server to send the response. Such scenarios though possible with Web Services, however, most of the current Web Services implementations support HTTP only with a minimal support and standardization for the other implementations.

Binary Interoperability: The mechanism can play an important role in those applications where performance and size of the serialized data is of critical importance. The binary interoperability methods serialize object from one platform into a binary stream, send them across the network and then de-serialize the same data on the other platform. Because both the parties address on the binary format to use, the serialization ensures that the binary data is successfully mapped to local data types for each platform.

The way that this option can be utilized for .NET-Java interoperability is by using the .NET Remoting as the binary protocol for communication. Bridges like JNBridge discussed earlier, provide the mapping of the Java objects into .NET Remoting and vice-versa. As the protocol uses a binary channel, the size of the packets going over the network is reduced and leads to a better performance than an approach using XML serialization. However, there are some disadvantages also associated with using the binary protocol. Applications using .NET Remoting typically live within the enterprise ad are rarely exposed to other organizations through firewalls or proxy servers (although .NET Remoting does support HTTP channel and SOAP formatter). The reason being that

the data types exposed by .NET Remoting server are based on the CLR, whereas the WS-I basic profile (and other Web Services implementations) rely on more standardized XSD style. In addition, using binary channel tends to enforce tight coupling with interfaces that are exposed, implying that if the methods of the exposed components change when using .Net Remoting, the stubs need to be re-generated and recompilation of the client and the server are required. In contrast, SOAP is more extensible; additional data can be included in the message header without having to modify the WSDL document.

Interoperability using CORBA: If binary interoperability is a must, but .NET Remoting is not a recognized standard within an organization, another alternative exists for organizations that have standardized on CORBA (Common Object Request Broker Architecture). The last twelve months have seen the introduction of a number of open source and commercial products that provide .NET clients with the ability to call and invoke remote objects written to the CORBA specification—and that aren't limited to interoperability only with Java-based applications. Typically, these products enable a .NET client to use IIOP (Internet Inter-ORB Protocol) to invoke remote components. One such commercial product is Borland's Janeva [BRI05]. This approach is most useful for organizations that have deployed CORBA server-side components but who do not wish to make any changes to them. Although using .NET Remoting provides binary interoperability, many toolkits still require some modification to server-side objects before clients can call the server component. Using a toolkit that allows a .NET client to natively reference a CORBA object can overcome this by requiring modifications only to the client.

One disadvantage is that the approach mandates ".NET client to Java server" style architecture. In many applications there are occasions where Java components need to invoke remote .NET objects—a good example of this is where an organization is implementing a Service Oriented Architecture (SOA) and has a combination of services developed using both technologies. For more information on achieving interoperability

between .NET and Java using CORBA and the product Janeva, the reader is referred to [BRI06].

Resource-Tier Interoperability: Interoperability solutions using Web services, .NET Remoting, or CORBA are typically synchronous in nature, and occur between two entities (a client and a server). In a typical scenario, the client calls the service, some data is returned after processing and the call in done. In applications that need to behave in a more asynchronous fashion, interoperability at the resource tier—which implies using either a database or a message queue—may be one option. Using a database or message queue to connect platforms based on .NET and J2EE can be one of the easiest ways to achieve interoperability between the two. Database solutions typically share a table between the two platforms, and each uses its preferred method for connecting to the database (ADO.NET for .NET, JDBC for the Java platform). Use of stored procedures at the database tier can also help in reducing duplication of code.

Interfacing with message queues works in a similar way, although each platform will normally have to obtain a driver from the message queue vendor in order to establish a connection. This may be a JMS (Java Message Service) driver for the J2EE platform, or a specific set of classes for .NET. For example, IBM offers WebSphere MQ 5.3 drivers with similar style interfaces for both Java and .NET. Many message queue vendors now also offer the ability to communicate via more standardized interfaces such as Web services.

In addition to providing a good support for asynchronous calls, using the resource tier can prove beneficial for n-n style interoperability, where multiple clients need to communicate with multiple servers. In the majority of the previous interoperability solutions, each client needs to know the location of each server. In a scenario using an intermediary database or message queue, although both sides must agree on the format of the message, the client does not necessarily need knowledge of the service location which

both negates the need for location-based information and can also help provide support for failover and load balancing.

A database or message queue form of communication protocol was however designed more specifically for asynchronous style communication. Using them in a synchronous style call (for example, where an ASP.NET page required a response before displaying the result to the user) can lead to potential performance issues. In addition, using either a database or message queue introduces yet another piece to the interoperability puzzle. For situations where many machines have a requirement for interoperability, it may be worth the investment in administration and additional machines; but in a scenario where just one client needs to interoperate with a single service, this solution just introduces potential overhead. For further reference, refer to the article in [BRI07].

Summary of the different methods: Each of the mechanisms discussed above for the .NET and Java interoperability have their own advantages and disadvantages. The incorporation of any of them in the existing applications or for building a distributed system requires taking into consideration the merits and the suitability of each of these methods. The following table lists some of the advantages and disadvantages of the approaches discussed above.

The interoperability mechanisms discussed in the above section are with a focus of discussing the model-related disparities that exists between component-object models, within the context of .NET and Java RMI. However, when integrating pre-existing components, there are more aspects to heterogeneity as well as integration than just technological differences. Communication between two components can be worded as a basic contract between the components. This contract needs to be more elaborated when additional requirements are imposed – such as security, transactions, etc. These details prove vital for the actual connection between components when the components form a part of a distributed system with certain performance and QoS constraints. Hence, such

details need to be importantly captured and in a way not visible to the component. These details comprise of the technology/middleware used to realize the connection, security issues such as encryption, quality of services, interface (semantic and signature) incompatibilities etc.

Table 3.1 Comparison of .NET-Java interoperability mechanisms [INT01]

	Integrated Platform Support	XML /SOAP Interopera-bility	Binary Interopera-bility	Designed for synchr-onous style calls	Designed for asyn-chronous style calls	Requires modifi-cations to existing server-side comp-onents	Extensible without re-compile
Web Services	Yes	Yes	No	Yes	No**	Yes***	Yes
.NET Remoting	Partial	Yes*	Yes	Yes	No	Partial	No
CORBA Interopera bility	Partial	No	Yes	Yes	No	No	No
Resource Tier	Yes	Optional	Optional	No	Yes	N/A	No
* Uses CLR data types, not XSD							
** Assuming HTTP as the transport							
*** Majority of modification are required to ensure							

These details are usually referred to as non-functional or extra-functional properties (NFPs) [BER03]. Reflecting these properties directly in the component's code can negatively influence the portability of the respective application across different platforms and middleware and hence, need to be incorporated outside the scope of the component. This problem is clearly reflected in one of the issues faced by the UniFrame Glue-Wrapper Generator Approach, which can be broadly classified into two categories:

- Provision of glue-wrapper code which can handle all the component interactions abiding to certain QoS metrics and other NFPs.
- Automation of the generation of this code to the extent possible.

The architectural primitives called “connectors” [BAL01] address the first issue to a large extent and hence the next section briefly introduces the concept of connectors and the background research in this field. The section also indicates a reference to the work which can form the basis of addressing the second issue.

3.4 Connectors

As mentioned earlier, connectors play a major role in component binding. The component models such as Java RMI, CORBA, .NET Remoting etc., used for Off-Shelf component development are mature enough for business applications; however they lack certain features which are significant in other distributed applications for different domains [BUL00]. For example, DCOM does not have any kind of architectural description, which prevents checking and simulation of a system without it being implemented. This makes DCOM an object-oriented middleware rather than a real component model. Also, all the component models mentioned above rely on a particular transport protocol. Neither of them employs a generic approach to component interactions. This drawback is eliminated by the use of Connectors. While components provide application-specific functionality, the connectors provide application-independent interaction mechanisms. This entity stands at the same level as components and mediates communication between components. It hides the technology used to implement the connection and makes the whole component system more flexible and tunable. It can wrap any of the interoperability mechanisms discussed in the previous section and allow further modifications or tuning to their behavior and operation - an aim which the UniFrame’s Glue-Wrapper Generator approach encapsulates.

Why do we need Connectors? Why can’t the interactions be modeled using the existing primitives such as components? Due to the limited amount of work done in this direction, there is often an inconsistent treatment and contradictory assumptions about Connectors. For example, connectors are often considered explicit at the level of architecture, but intangible in a system’s implementation. Also, due to the fact the

differences between components and connectors are very subtle, it is tempting for a developer to use components themselves to model component interactions if needed. Section 3.4.1 discusses some of the major issues that motivate the use of connectors as first class entities in software architectures [PER92, SHA96]. Section 3.4.2 then introduces a theoretical implementation model for Connectors [BUL00], which seems the most appropriate from the stand-point of glue-wrapper code functionality.

3.4.1 Motivation for the Use of Connector Architecture

Deployment Anomaly: the deployment anomaly is first discussed in [BAL01]. The problem is inherent to distributed systems and remains one of the strongest motivation factors for introducing connectors as a first class entity in component-based software architectures. The problem is explained with the help of the following figure.

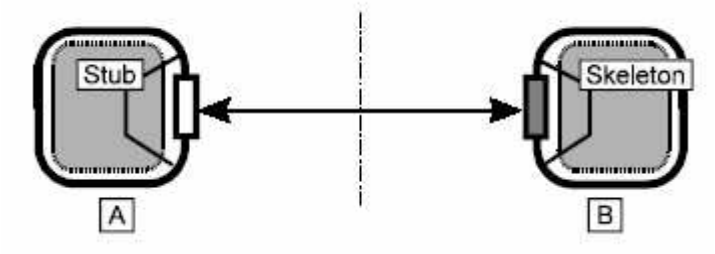


Figure 3.1 Server and client across distribution boundaries [BUL00]

The above figure represents two components, which are autonomous and completely contained in their own grey boxes and have interfaces shown by the rectangular boxes at their boundary.

Most middleware platforms create proxy objects which mediate the communication, the common ones being *stub* at client side and *skeleton* on server side. Since different types of middleware are not mutually compatible, neither during development process nor during runtime, the middleware platform for given application

must be chosen in early stage of application development and the implementation code must reflect the needs of the selected platform.

The code for implementing the stubs/skeletons and other middleware specific code are contained within the trapezoidal boxes within the components. These symbols thus represent the middleware dependency of the implementation code of the component. When a need arises to change the underlying middleware, the code of the component will need to be changed to adapt to the new platform. However, source code to the commercial components is not freely available and makes such a process impossible and besides will also be a very time-consuming process. An obvious solution to this problem seems the use of a component that can mediate the communication between components A and B. Such mediation can be depicted by the following figure:

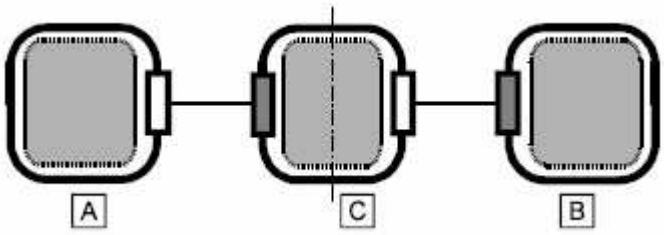


Figure 3.2 Component modeled as a connector – distribution boundary across the code [BUL00]

The implementation code of this component would be generated to mediate between these two specific components; whether it is generated manually or in an automated fashion by the Glue-Wrapper Generator, is less irrelevant here. However, the distribution boundary in this case will cross the component C from within the implementation code rather than the interface ties. At the level of architecture description, the distribution boundary can only cross a compound component at its interface ties and not implementation code. A connector resolves the problem by the mere definition of it and its architectural description as a first class entity, just like a component. The

connection in the above figure would be modified as below with the intervention of a connector:

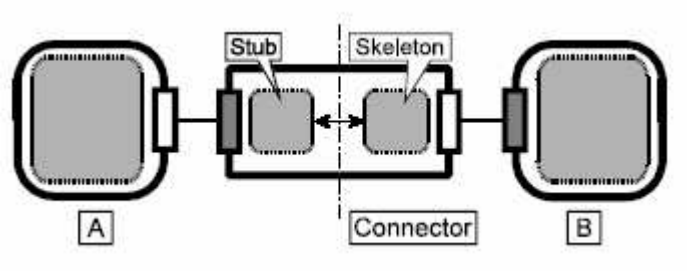


Figure 3.3 Connector mediating the communication [BUL00]

In the above figure, the connection between components A and B is mediated between the Connector; top-level entity which is a part of the architectural description of the system consisting of A and B. The implementation of the connector is inherently distributed and consists of parts, which are locally attached to components participating in the interaction the connector mediates. The distribution boundary is clearly defined therefore the connector does not suffer from the problem mentioned above where the distribution boundary was crossing the component's implementation code.

Connector Lifecycle: The life-cycle of a component differs significantly from that of a connector. Off-The-shelf components are supposed to be pre-fabricated building blocks with specific functionality and a set of parameters, which can be used to set or adjust its non-functional properties. When the development of a component is complete, the source code of a component implementation is compiled into binary form, which is then packaged and prepared for distribution to component users. Developer of a component application with a need for specific functionality can then obtain a component in packaged form, deploy it into a deployment dock and after satisfying its requirements, the component can be instantiated and run. Except for the configuration of its non-functional properties, no modifications need to be made to the component implementation for it to run in the deployment dock. If for example, the underlying

middleware platform is changed for another with better or more suitable properties, a component should not require any changes be made to it.

Such requirements conflicts with the concept of connectors, since the goal is to allow the selection of the middleware platform and the execution environment as late as during the deployment of component application, or in other words a distributed system in reference to UniFrame. Therefore, the implementation of a connector has to be designed so that it can be adapted or generated according to the choice made by the deployer; in case of UniFrame, it would be the choice made by the System Generator in an automated fashion. Each part of a connector is generated by the deployment dock hosting the component it is attached to. The resulting implantation code is then compiled and loaded into the deployment dock runtime, where it can be instantiated and bound to the component interface participating in the interaction.

When using connectors to mediate component interactions, the element that remains in composing an application is setting the interfaces and fine tuning them according to the specifications of the application to be composed. The implementation details of the component interactions according to the technology need to be only written once and then reused according to the different components and their use within the applications. These implementation details can be stored in the form of a knowledge base known as the "technology knowledge-base" and include details in relevance to the component models supported by the glue-wrapper generator.

Platform Dependency: The use of connectors allows for a greater flexibility when choosing a transport method appropriate for a specific interaction. This allows the Glue-Wrapper Generator to decide an appropriate interoperability mechanism, as discussed before, to be incorporated within the connector architecture. As already indicated in the previous section, the implementation code of a connector depends on the underlying middleware platform and the execution environment. In addition, an application may require the interactions to meet certain criteria concerning their non-functional properties,

such as memory consumption, performance, reliability or quality-of-service guarantees. The Glue-Wrapper Generator could generate the glue-code based on the connector architecture and the QoS specifications of the connectors between a set of two components could be based on the QoS composition-decomposition model defined in [SUN03]. These specifications form an important part in system composition to meet the QoS requirements specified in the system query in the first place.

Most of these criteria are mutually exclusive and therefore, it is feasible to have multiple connector architectures for single connector type, each of them putting emphasis on different factor. The choice of the connector architecture, underlying middleware and the execution platform need to reflect the actual needs of an application or the criteria requirements put on the system as a whole. Hence, using connectors to mediate component interactions allows the UniFrame GWG to make such decisions as late as during the deployment of the entire system, without affecting the application components.

Connector architecture also allows for the capture of the nature of interactions, which can then be used to generate effective connector implementations for specific types of interactions. For example, when modeling an unreliable channel for streaming multimedia, using a remote procedure call provided by an advanced middleware (e.g., CORBA) is clearly not needed, when such functionality can be gained by using UDP datagram service or UNIX pipe (in case the connected components are located on the same node). Again, when two components need to communicate by sending thousands of messages within a second, communication using a heavy protocol such as SOAP and XML could result in a large overhead leading to a delay in achieving the necessary results from the communication.

There are different models proposed for the connector implementations. The work done by Balek and Plasil [BAL01] combined with the work of Bulej and Bures [BUL00] is now presented below. The work together gives one of the theoretical implementations

for the connector model suitable for a generator that could automate the generation of connectors based on this model.

3.4.2 Connector Model

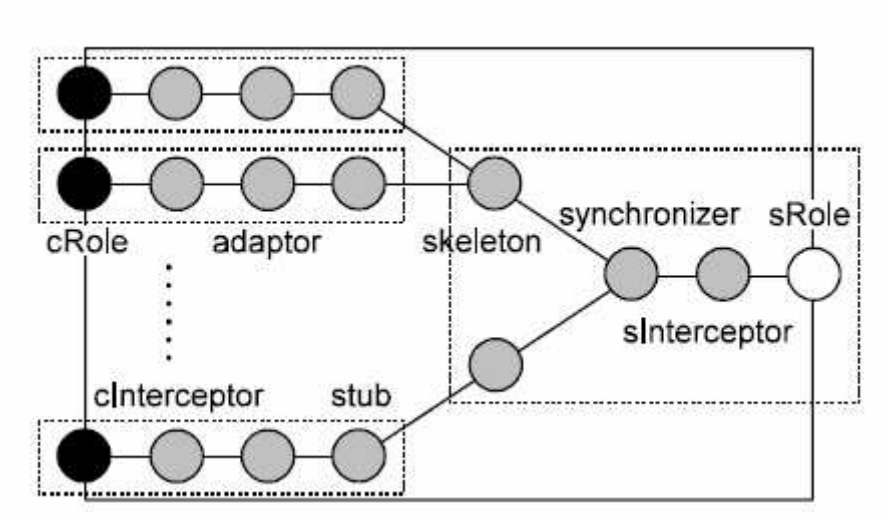


Figure 3.4 Connector Architecture and Frame

The Connector model consists of:

Connector Frame: In [BAL01] the connector structure is proposed as a hierarchical model, which reflects the top-down approach often used in design and development of component applications. The Frame is the topmost abstraction in connector design and models a connector as a black-box entity. Its purpose is to allow developers of component applications to work with various types of connections (procedure call, message passing, pipe etc.) without explicit knowledge of their implementation. A detailed taxonomy of connectors can be found in [MEH00]. The black rectangle in the above figure depicts a connector frame. Connector roles (dotted rectangles) serve as attachment ports and their cardinality determines the number of components allowed to connect to a particular port. The above example depicts a connector with a single server role (sRole) and multiple client roles (cRole). Consistent with the structural model of interactions, a role needs to have its provisions and

requirements defined. These are directed interfaces, i.e., interfaces which a role either provides a binding to (full circles) or requires to bound to (empty circles). The cardinality of provisions and requirements is used to specify whether an interface is optional or mandatory and whether it is singleton or comes in multiple instances. In the above example, all role provisions and requirements are mandatory singletons.

Since the definition of a connector has to be machine readable, the designer of a connector type is required to write the connector definition in a language with support for connectors. In [BUL00], the Component Definition Language (CDL) [MEN98] has been used for the purpose. The reference does not bind itself to any particular specification language and chose CDL for the supporting prototype. For the purpose of UniFrame, these specifications could be well supported by TLG (introduced in Section 2.1.2).

Connector Architecture: The level of details provided by the description of connector frame is sufficient for connecting components together using various connector types. However, the definition of the frame does not say anything about the internal details of the connector. The Connector Architecture provides these internal details about the functionality hidden by the black-box view, such as logging, transport security, interface adaptation, synchronization, etc. It can be called as the gray-box view of the connector and allows for the description of connector internals in the form of composition of primitive elements [BUL00], the building blocks of connector implementation. The gray circles inside the solid rectangle (connector frame) represent the primitive elements. Each element provides the connector implementation with specific functionality and is further indivisible. In the above figure, the client-side architecture is shown to be composed of the following primitive elements:

- cInterceptor: can be used to log incoming method invocations.
- adaptor: can be incorporated if interface adaptation is required.
- stub: provides the functionality of remote procedure call client, such as marshalling, sending a request over the network to the server and receiving and unmarshalling the reply.

The server side is composed of:

- skeleton: provides the remote procedure call server (opposite of stub).
- synchronizer: can be incorporated inside the connector if the requests at the server need to be serialized or implementing a threading model.
- sInterceptor: primitive element used for logging on the server side.

The dotted rectangles mark the boundaries of the connector units. Connector unit groups together elements to be instantiated in the same address space. The links between elements sharing the same address space are created by an entity called "connector builder" in [BUL00]. Such an entity could be the part of the UniFrame GWG which can establish the links between the primitive elements after the knowledge is supplied to the builder from the technology and other QoS descriptions of the connection. The responsibility of establishing links across unit boundaries is delegated to the elements on both sides of the link. Typically, these links use the underlying middleware to implement the link, but can also be implemented by the elements by themselves. The framework hence allows for the encapsulation of different kind of interoperability mechanisms for the generation of connectors. Moreover, the use of connector units allows the expression of the inherent distributed nature of connectors, discussed earlier.

Thus, while the connector frame defines a basic connector type, connector architecture describes the functionality of the connector. Since, there could be several ways to implement a given connection type, therefore there could be several architectures implementing a single frame. The figure above depicts just one such architecture. The decision could be based on the communication pattern desired between the two components on the basis of the NFP constraint on the connection.

The [BUL00] also specifies a generator framework which allows for the semi/fully automatic instantiation of the connectors based on the model above. The framework is based on the life-cycle of the connectors and leads to an automatic generation of the connectors based on certain inputs from the user. The assumption is that

the specifications deciding the nature of the connectors to be generated exists and does not attempt an answer to it. It addresses automation beyond this point. The issue however can be addressed with the help of the GDM of the UniFrame approach which can decide the QoS constraints on the connectors mediating between the components based on the QoS composition/decomposition model and the nature of the connectors based on the requirements of the system under consideration. The connection is attempted and proposed in the Chapter 6 with respect to the generation of connectors for connecting heterogeneous discovery services. In addition, the validation of the model in [BUL00] is carried out with respect to Java RMI and CORBA components using the same language, Java. An extension of the work consists of applying the principles of connector generation to a non-Java language based component model such as .NET. Chapter 6 provides for such an approach targeting .NET and Java RMI. However, the design of the generator is out of the scope of the thesis.

The chapters till now have discussed the .NET component model from the perspective of seamlessly encompassing the components developed using this model under the framework of UniFrame. Chapter 2 introduced the framework and outlined some of the interoperability bridges between components of .NET and Java component model. Chapter 3 addressed the problem of heterogeneity from a similar perspective and also mentioned Web Services as one of the interoperability mechanisms. However this gives rise to one question – can .NET only be incorporated as a part of the UniFrame? Could the two frameworks also complement each other? What aspects need to be considered while finding a solution to these questions? The following chapter is an attempt in this direction.

4 UNIFRAME IS NOT WEB SERVICES – AN ANALYSIS

[MIC04] defines .NET as: "*Microsoft® .NET is a set of software technologies for connecting information, people, systems, and devices. This new generation of technology is based on Web services—small building-block applications that can connect to each other as well as to other, larger applications over the Internet.*" Thus, it is evident that Web Services form one of the inherent features of the .NET framework. The main components of the Web Services, namely, SOAP, WSDL, UDDI and XML, were discussed in Chapter 2 and Chapter 3 also described Web Services as an interoperability mechanism for different component models. However, with a whole set of standards associated with the Web Services, they can also be considered as a framework that allows developing component-based software solutions for distributed systems. The advantages of distributed computing resources (as discussed in Chapter 1) are significant and have necessitated the availability of such frameworks that could facilitate the efficient integration of distributed resources. Innovations in this field have led to the development of other such frameworks as Enterprise Application Integration (EAI) solutions, Business Integration (BI) Solutions, Open Grid Services Architecture (OGSA), and UniFrame. Defining the role of Web Services in this manner raises a question – are UniFrame and Web Services based on the same principles and do they achieve the same end-result? These questions are often manifested as misconceptions leading to an ambiguous understanding of each of the frameworks. Web Services form an integral part of the .NET distributed paradigm. Since, the goal of the thesis is to analyze the two paradigms of .NET and UniFrame with a synergistic approach exploring if there are any possibilities where the two paradigms can complement each other or what are the characteristics of the .NET component model which the UniFrame approach can subsume or extend;

answering these questions becomes an important aspect of the thesis. Hence, as per the focus of the thesis, it becomes important to provide a clear understanding of this facet of .NET, namely Web Services in the context of UniFrame. This chapter accomplishes this task, namely – analyzing the Web Services and UniFrame paradigms [GUP03].

At the basic level, any paradigm that entails an ensemble of heterogeneous components has the following characteristics:

- Development framework for Composable components.
  - Whether any special technique/tool is required or components could be developed on any computing platform.
  - Kind of components supported by the architecture (hardware resources/ software resources/ object model-based components).
  - Description about the components.
- Publishing of components on the network.
  - Process of making the components known to others.
- Discovery and composition of components.
  - Mechanism to discover and integrate the selected components.

All the above characteristics are evident in both the UniFrame and Web Services paradigm. However, there are certain details which make a difference in the way the end-result is achieved using these two paradigms. Hence, to provide a comprehensive analysis of the two paradigms, the comparison proceeds in two folds, a) Architecture-based Comparison and b) Model-based Comparison. Architecture-based comparison outlines the differences in the characteristic features of the fundamental architecture of the two frameworks in terms of the basic underlying principles. Model-based comparison focuses on the differences that are evident in the model employed by the architecture at a more abstract level. Comparisons at both the levels are of significance for a detailed evaluation methodology. The next section outlines the architecture-based comparison of the two models and Section 4.2 outlines the model-based comparison.

4.1 Architecture-based Comparison

Table 4.1 indicates the detailed comparison, from the perspective of the underlying architecture, for both the approaches. The metrics used for the comparison are based on the above mentioned desirable characteristics of any component-based framework.

Table 4.1 Architectural comparison of Web Services and UniFrame paradigms

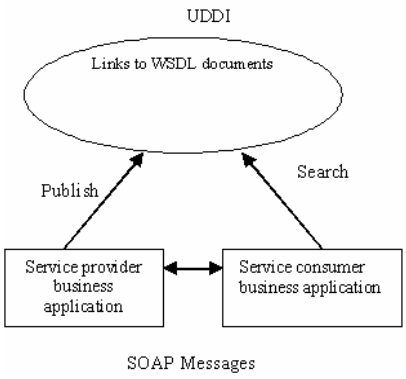
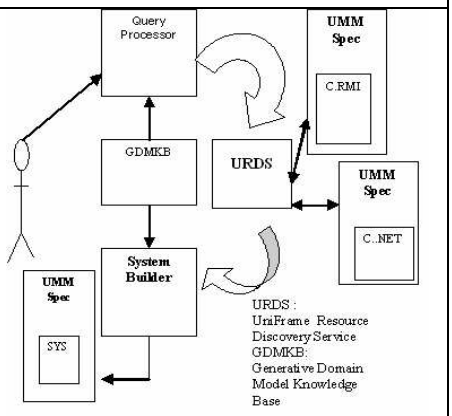
	WEB SERVICES FRAMEWORK	UNIFRAME
OBJECTIVE	To provide a set of related standards which allow building of dynamic, loosely coupled systems composed of services, not bounded to any implementation and can be published, described, located and invoked over a network, more generally World Wide Web.	To create a comprehensive framework that unifies the existing and emerging distributed component/service models under a common meta-model, that enables the discovery, interoperability, and collaboration of components via generative software techniques.
GENERAL FRAMEWORK/ ARCHITECTURE	 Figure (a) Web Services's overall process	 Figure (b) UniFrame's overall process
OVERALL PROCESS	▪ Service Development and Deployment (leveraging all different platforms to one standard of Web Services) using different Web Services development tools and software provided by vendors. ▪ Formal description of services (WSDL). ▪ Registration of services with UDDI (publish).	▪ Developing components using a specific component model (DCOM/RMI/CORBA/.NET/Web Services) and associated UMM specifications. ▪ Querying the UniFrame for creating a system with desired Quality of Service parameters. ▪ Select, out of the discovered components.

Table 4.1 Continued

OVERALL PROCESS Continued	▪ Discovery of services (Find) using the registry API – directory service. ▪ Binding with the Service (Bind)	▪ Compare to see if the test results with the discovered components satisfy criteria. ▪ Refine Query or select alternate components to re-build the system.
SERVICE/ COMPONENT DEVELOPMENT & DEPLOYMENT	▪ Development using frameworks that support them (e.g. .NET) or using different object models, which are then leveraged as services using the toolkits that support the technology. ▪ Registering Services with the UDDI public/private registry.	▪ Components are developed using inherent mechanism. ▪ Deployment also under the same model with extra infrastructure provided by UniFrame to support seamless interoperation and system generation.
DESCRIPTION OF COMPONENTS/ SERVICES	Web Service Description Language Document (WSDL file – XML).	UniFrame Meta-Component Model Description (UMM Specifications – informal text and XML)
DISCOVERY	Discovery through the UDDI Business or private registries (static registries)	Discovery through a search process involving active entities – headhunters and active registries [UniFrame Resource Discovery Service (URDS) Framework].
INTEROPERA-BILITY OF SERVICES/ COMPONENTS	XML (standard for data exchange) and SOAP (Simple Object Access Protocol).	Automatic generation of glues and wrappers.
SYSTEM INTEGRATION	▪ A hand-crafted approach wherein the responsibility of integration lies with the application developer by means of APIs of the Web Services. ▪ Need to incorporate Web Services interfaces and integration capabilities within the existing "application integrating" tools and products.	A comprehensive model-based approach forms the backbone of the system integration process right from the initial stages. The model follows an architecture-centric, domain-based and a technology-independent approach. The process may be manual, completely automatic or a mix of both.
RELIABILITY OF COMPOSED SYSTEM	Reliance on a third party (Web Service Auditors) which guarantees the reliability of a web service on basis of testing and certification during its creation as well as operational stage.	Reliability based on test cases and formalism and a strong mathematical foundation of event traces and two level grammar.
ADVANTAGES	▪ Builds upon open text-based standards (XML), thus aiding in interoperability. Contd…	▪ No requirement of additional software tool to build components. ▪ Automatic generation of glues and wrappers. ▪ Quality of Service validation and assurance through event traces and formal domain knowledge; backed by a mathematical foundation.

Table 4.1 Continued

ADVANTAGES Continued	▪ Less additional cost involved in adoption, since employs existing infrastructure (Internet) and applications can be repackaged as Web Services.	▪ Use of aspect-oriented programming to weave in the notion of QoS into the framework distinguishes UniFrame. ▪ Active search process involving the notion of "headhunters".
LIMITATIONS	▪ Relatively new; standardization in progress, hence, Web Services created with current tools will not be compatible with the future technologies. ▪ Use of text-based standards, XML, for communication may affect performance in some critical. Applications. ▪ No standardized methods devised for assuring and validating Quality of Service; Use of third party "web service auditors".	▪ No standardization reached yet. ▪ Experimentation and performance evaluation at a large scale and in a realistic domain not complete.

As is evident from the tabulated comparison, both UniFrame and Web Services build on the notion of Service Oriented Architecture. However, based on their different standpoint, the notion of service in each case has different implications. For the Web Services paradigm, a service is "any software component which is accessible through standard web protocol [THU01]". UniFrame defines its concept of service as "an intensive computational effort or access to underlying resource [RAJ00]". While Web Services attaches a set of technology specifics in its point of view of a service, UniFrame adopts a very generic view. Thus, results the difference in the way each paradigm handles their integration.

In case of Web Services, the integration platform is provided by means of standardization of four basic parts of an integration procedure, namely, a) Representation and transfer of data, b) Extensible message-processing format, c) Service description language, and d) A way to locate services. The general architecture under which integration of heterogeneous business processes in carried out using Web Services is based on these standards and can be depicted with the help Figure (a) in Table 4.1. Service Publishers register their web services and their descriptions with multiple operator nodes on the network that provide a cloud of internet-wide repositories of Web

Services metadata. These nodes implement the UDDI specifications and are publicly available. The UDDI directory exposes a set of APIs in the form of a SOAP-based Web Service, both for the publishers and the requestors of services. By the means of these APIs, the clients can locate the service that meets their requirements. The protocol employed is the DISCO (Discovery) Specification that defines an algorithm that enables the client to locate the service descriptions of the service. If the Web Service Client knows the location of the service descriptions, the above discovery process could be bypassed. Once the location of the service description is known to the client, the client makes a judgment of the services offered by the client and the mechanisms to invoke it. This description could be utilized in the generation of a proxy of the remote service and hence aids the developer to include the functionality of the service in its application. The functionality of the service could be invoked using the standard wire-formats of the web services, namely XML data structures over HTTP. In a similar manner, an application could incorporate the services of multiple web services which it discovers over the network. A method is invoked on this service and the results are then obtained back after the necessary processing. The result is a distributed application employing a more or less point-to-point communication from the perspective of the client that carries out the necessary integration to construct the system at hand. Integration procedure follows a hand-crafted approach based on the decisions of the client for the services to be integrated. To summarize it, Web Services are a collection of technology standards which by themselves, do not constitute a service-oriented architecture, but only enable it.

On the other hand, the UniFrame's approach is to provide a comprehensive framework for the software realization of a DCS, which aids in its design, taking into account all the challenges associated with the inherent features of a DCS such as heterogeneity, local autonomy and open architecture. The overall process of system assembling under the UniFrame approach is discussed in detail in Chapter 2, Section 2.1. Figure (b) of Table 4.1 also depicts this overall process. UniFrame's approach is centered around the existence of an extensive knowledgebase for a GDM that provides the basis for constructing DCS solutions over problem space. The knowledgebase is constructed by

business domain experts who perform requirement analysis for a particular domain and model the business context in that domain for which the DCS is to be constructed. They derive the Business Reference Models and place them in the UniFrame's knowledgebase which define space of problems they can solve. Components are developed by developers in their choice of component models using the native development techniques. Their basic properties are then specified according to the standards established by the UMM. Several Business Reference Models can share one component. The UniFrame system is queried for a desired DCS. Based on the query, Business Reference Model is identified along with the abstract definitions of the components that satisfy the Business Reference Model. Concrete implementations for these abstract definitions are discovered from the components deployed on the network. System is generated out of these components and validated for its QoS against the given query requirements. If the system does not meet the specifications, the process is repeated or the requirements are refined till a satisfactory result is achieved.

Thus, though the objective of the two paradigms is the same, the underlying approach differentiates them in terms of the process involved and the DCS constructed. The above mentioned overall approach can be further analyzed by identifying and evaluating different architectural metrics associated with the two paradigms. The following sections now present a detailed description these fundamental metrics. These are discovery services, service descriptions, registration mechanism and quality of service assurances entailed by the two paradigms.

4.1.1 Discovery Services

Web Services Discovery Process: The term discovery refers to the process of locating "Web Services" by means of registries. This process is carried out by businesses searching for services offering specific functionalities. Web Services Registries and Brokerages facilitate the discovery process and enable interactions between the service

providers and requesters. The discovery process is classified into two categories [GUP03]:

- *Direct Discovery:* This kind of discovery involves obtaining data from a registry, which is maintained by the service provider itself. It provides an advantage to the service requester of the data having a higher probability of being more accurate and recent.
- *Indirect Discovery:* The Service requester obtains data about a Web Service from a registry, which is maintained by a third party organization. In this case, a service requester has the facility to evaluate a number of Web Services before deciding on any particular one. However, the freshness of the information available to the service requesters depends on the frequency with which it is updated in these registries.

A service provider publishes the WSDL document containing the description of its Web Service, with the UDDI, which makes locations of such WSDL files available to a service requester. The Service Requester searches the UDDI based on certain criterion, such as functionality or a Quality of Service (QoS) attribute. Once it discovers a service, meeting its needs, it knows the method of accessing the Web Service by means of the WSDL file. It can now communicate with the Web Service directly via SOAP messages.

There are a few other discovery technologies, which support the discovery of Web Services apart from the UDDI specifications – ebXML [EBX] for example. ebXML stands for "e*lectronic business XML*" and is an electronic business standard sponsored by UN/CEFACT (United Nations Center for Trade Facilitation and Electronic Business) and OASIS (Organization for Advancement of Structural Information Standards). It defines a standard that allows services to find each other and conduct transactions based on well-defined XML messages within the context of standard business processes which are governed by standard or mutually-negotiated partner agreement. The ebXML provides standards for specification of business processes, repositories for other services to discover them and a mechanism for services to encompass them in order to support an application (by means of a common transport mechanism for exchanging messages

between organizations). In this context ebXML also falls in the category of the UniFrame and Web Services paradigm. In addition, the ebXML standards also support the notion of Web Services in all the 3 respects:

- services specifications (Web Services - WSDL, ebXML – Collaboration Protocol Profile).
- publication and discovery (Web Services – UDDI, ebXML – ebXML Registry Services).
- invocation (Web Services – SOAP and HTTP, ebXML – ebXML Messaging Service based on SOAP and HTTP).

A service developer/organization can combine these technologies with the Web services in order to take advantage of the features of both. For example, UDDI currently does not support a security model whereas ebXML does as part of its ebXML Messaging Service and so an organization can advertise its services through UDDI, on the other hand store its trading agreements and contracts through ebXML.

However, in both the cases, whether it is UDDI or through the ebXML Registry Services, the general procedure for discovery of Web Services is as depicted in the Figure 4.1. The basic underlying concept is: Web Services register their specifications in a repository which are then searched by the interested clients. After retrieving the Web Services specifications in which the client is interested in, the client can now invoke the Web Service.

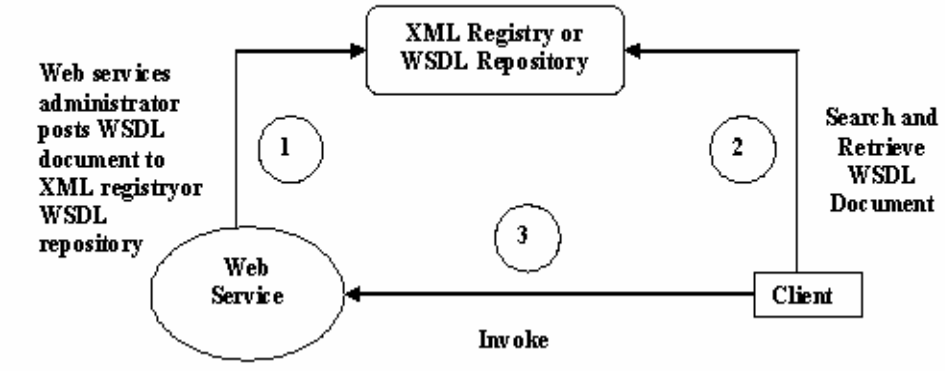


Figure 4.1 Discovery of Web Services

UniFrame Resource Discovery Service (URDS) Framework: The URDS framework [SIR01] supports the notion of automated discovery process of the UniFrame paradigm wherein new services are dynamically discovered while providing the clients with a directory-style access to the services. The services are searched based on their quality requirements within an "administratively-scoped" discovery process. In other words, the URDS framework locates services within an administratively defined logical domain – in UniFrame a domain refers to industry specific markets such as Financial Services, Medical domain and Manufacturing Services, etc. The URDS infrastructure consists of the following parts: (a) the Internet Component Broker (ICB), (b) Headhunters (HH), (c) Meta-Repository (MR) and (d) Active Registries (AR). All these entities are arranged in a hierarchical structure making the discovery service scalable allowing extending the search scope of the URDS by tuning the number of the different entities participating in the discovery process. The scalability of the URDS framework is further discussed in Chapter 5.

The ICB is the UniFrame's analogous entity to the conventional request broker in other architectures. However, in addition to performing the functions of a conventional broker, it also performs other functions such as, it ensures the authentication of the principals of the system (namely the Headhunters and Active Registries); cooperates with other ICB's deployed on the network to provide matchmaking between service producers and consumers; and acts as a mediator between two components adhering to different component models. The entities which constitute the ICB have been explained in detail in Chapter 5. A Headhunter is equivalent to a binder or trader in other models. However, unlike the trader, here the onus of registering components lies with the headhunter and not on the components themselves. Hence, the headhunter is capable of detecting the presence of service providers on the network, register the functionality of these service providers and return a list of service providers, which matches the requirements of the consumer requests forwarded by the Query Manager, to the ICB. The services are discovered by means of Active Registries (discussed later), with which the services are registered. The discovery process employed could vary from standard search techniques

such as lookup discovery to broadcasts and multicasts to specific machines. What distinguishes the URDS from a majority of the distributed discovery services proposed and implemented by the industry and academia (some of them being Archie, Jini, CORBA trader, Ninf etc.), is that the URDS architecture spans across heterogeneous component models enabling the discovery of components developed on different paradigms such as .NET, Java RMI, Web Services, etc. It also acts as pre-cursor in enabling their participation in composing a heterogeneous distributed system. This is done by enabling the search based on the UMM specifications (discussed next) of the components and with the support for the discovery of "Adapter Components" which participate in the generation of the glue-wrapper code for composing a heterogeneous system. The main characteristics of the process under the two paradigms have been tabulated in Table 4.2.

Table 4.2 Discovery process under Web Services and UniFrame paradigms

DISCOVERY PROCESS		
	Web Services	UniFrame
Characteristics	▪ Process of locating Web Services to meet specific needs, through registries ▪ Activated and Executed by Service Requestors	▪ Process of locating components satisfying the QoS parameters specified in the query ▪ Activated by a "system" query and automatically executed by headhunters-active registries

4.1.2 Service Descriptions

Web Service Description Language (WSDL) Document: It is an XML document for describing Web Services as a set of endpoints operating on messages containing either document-oriented (messaging) or RPC-payloads. Service interfaces are defined abstractly in terms of message structures and sequences of simple message exchanges and then bound to a concrete network protocol and data-encoding format to define an end-point. Related concrete end-points are bundled to define abstract end-points (services). The WSDL is extensible to allow description of end-points and the concrete representation of their messages for a variety of different message formats and network

protocols [DIE03]. The WSDL file has five primary elements, which are used to describe a Web Service and appear in the WSDL file in the following order:

1. <types> element : defines the various *data types* in exchanging messages.
2. <messages> element: describes the *messages* being communicated.
3. <portType> element: defines a *set of operations* and the methods associated with those operations.
4. <binding> element: specifies *protocol for various operations* and describes how to map the abstract content of the messages to a concrete format.
5. <service> element: groups a *set of related ports* (operations) together; specifies the actual location (URL) of the Web Service on the server.

Figure 4.2 shows a WSDL document for a component of the type "CashierValidationServer". The service is named as "Cashier Validation Service". The different nodes of the file as discussed above can be seen for this service in the figure. The five elements mentioned which make up a WSDL file can be grouped into the following two categories: 1) Abstract Definitions consisting of the nodes, types, messages, and port types, and 2) Concrete Definitions comprising the bindings and the services nodes. The abstract sections define SOAP messages in a platform- and language-independent manner; containing no machine- or language-specific elements. This gives an abstract definition to a set of services that diverse Web sites can implement. Site-specific matters such as serialization are relegated to the concrete descriptions. The WSDL document for the Cashier Validation Service supports two roles in its interfaces, "Validate User" and "Grant Access". As an example, consider the *type* elements for "ValidateUser" which defines this method with its corresponding data types – (<s:element minOccurs="**1**" maxOccurs="**1**" name="**userID**" type="**s:int**" /> , <s:element minOccurs="**1**" maxOccurs="**1**" name="**passwordID**" type="**s:int**" />). This shows that the ValidateUser accepts parameters userID and passwordID as int datatypes. The next set is the <message> elements comprising the Messages section. If operations are considered as functions, then a <message> element defines the parameters to that function. Each <part> child element in the <message> element corresponds to a parameter. Similarly, other

details of the elements, map the service interface specifications in terms of the port, binding and service elements in the manner described above.

```
<?xml version="1.0" encoding="utf-8" ?>
        <definitions xmlns:http="ttp://schemas.xmlsoap.org/wsdl/http/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:s="http://www.w3.org/2001/XMLSchema"
        xmlns:s0="http://WebSvc.XT.Local/CashierValidationService"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        targetNamespace="http://WebSvc.XT.Local/Banking"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
         <s:schema elementFormDefault="qualified"
        targetNamespace="http://WebSvc.XT.Local/Banking">
        <s:element name="ValidateUser"> <!--METHOD 1 of Web Service-->
        <s:complexType>
        <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="userID" type="s:int" />
        <s:element minOccurs="1" maxOccurs="1" name="passwordID" type="s:int"/>
        </s:sequence>
         </s:complexType>
        </s:element>
        <s:element name=" ValidateUserResponse">
        <s:complexType>
        <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name=" ValidateUserResponseResult"
        type="s:string" />
        </s:sequence>
        </s:complexType>
        </s:element>

        <s:element name="GrantAccess">
        <s:complexType>
        <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="intCashierID" type="s:int" />
        <s:element minOccurs="1" maxOccurs="1" name="intRoleID" type="s:int" />
        </s:sequence>
        </s:complexType>
        </s:element>
        <s:element name=" GrantAccessResponse">
        <s:complexType>
        <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="GrantAccessResult" type="s:string" />
        </s:sequence>
        </s:complexType>
        </s:element>
        </s:schema>
 </types>

<message name="ValidateUserSoapIn">
        <part name="parameters" element="s0: ValidateUser" />
        </message>

        <message name="ValidateUserSoapOut">
        <part name="parameters" element="s0: ValidateUserResponse" />
</message>
```

Figure 4.2 WSDL description for a Cashier Validation Service

```
        <message name=" GrantAccessSoapIn">
        <part name="parameters" element="s0: GrantAccess" />
        </message>

        <message name=" GrantAccessSoapOut">
        <part name="parameters" element="s0: GrantAccessResponse" />
         </message>

<portType name="ValidationSoap">
    <operation name=" ValidateUserResponse">
          <documentation> ValidateUserResponse(int brandID, int emailID) - Uses the spam
        filter to check again an email.</documentation>
          <input message="s0:ValidateUserSoapIn" />
          <output message="s0:ValidateUserSoapOut" />
   </operation>
   <operation name="GrantAccess">
          <documentation>GrantAccess(int intCashierID, int intRoleID) - Checks the substitution
        strings in an email .</documentation>
          <input message="s0:GrantAccessSoapIn" />
          <output message="s0:GrantAccessSoapOut" />
  </operation>
</portType>

<binding name="CashierValidationServiceSoap" type="s0: CashierValidationServiceSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
        <operation name="ValidateUser">
        <soap:operation
  soapAction="http://WebSvc.XT.Local/CashierValidationService/ValidateUser"
        style="document" />
        <input>
                <soap:body use="literal" />
          </input>
        <output>
          <soap:body use="literal" />
        </output>
    </operation>
   <operation name="EmailValidator">
        <soap:operation
  soapAction="http://WebSvc.XT.Local/CashierValidationService/GrantAccess"
        style="document" />
        <input>
                <soap:body use="literal" />
         </input>
        <output>
                <soap:body use="literal" />
        </output>
     </operation>
</binding>

<service name="CashierValidationService">
<port name="CashierValidationServiceSoap" binding="s0: CashierValidationServiceSoap">
        <soap:address
        location="http://www.BankingServices.com/CashierValidation/CashierValidationService
        .asmx" />
  </port>
</service>
  </definitions>
```

Figure 4.2 Continued

UniFrame Meta-Component Model (UMM) Description: The discovery process outlined in the previous section mentioned that the headhunters enable the discovery of components on the basis of many factors including the quality requirements. As mentioned in Chapter 2 (Section 2.1 - Introduction to UniFrame), the components under the paradigm of UniFrame conform to a standard abstract component model. The UMM specifications follow a one-to-one correspondence with the attributes of the components as described in the abstract component model. Figure 4.3 depicts the informal specifications of one such component, CashierValidationServer, which forms a part of the abstract component model for an account management system in the domain of Banking. The component

Abstract Component: *CashierValidationServerCase1*
1. Component Name: *CashierValidationServer*
2. Component Subcase: *CashierValidationServerCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide Cashier validation service in banking.
6. Computational Attributes:
    6.1 Inherent Attributes:
        6.1.1 id: N/A
        6.1.2 Version: version 1.0
        6.1.3 Author: N/A
        6.1.4 Date: N/A
        6.1.5 Validity: N/A
        6.1.6 Atomicity: Yes
        6.1.7 Registration: N/A
        6.1.8 Model: N/A
    6.2 Functional Attributes:
        6.2.1 Function description: Act as validation server for Cashiers in banking.
        6.2.2 Algorithm: N/A
        6.2.3 Complexity: N/A
        6.2.4 Syntactic Contract
            6.2.4.1 Provided Interface: *IValidationCase1*
            6.2.4.2 Required Interface: NONE
            6.2.5 Technology: N/A
            6.2.6 Expected Resources: N/A
            6.2.7 Design Patterns: NONE
            6.2.8 Known Usage: NONE
            6.2.9 Alias: NONE
7. Cooperation Attributes
    7.1 Preprocessing Collaborators: *CashierTerminalCase1*
    7.2 Postprocessing Collaborators: NONE

Figure 4.3 Informal representation of the UMM specifications of a component

8. Auxiliary Attributes:
    8.1 Mobility: No
    8.2 Security: *L0*
    8.3 Fault tolerance: *L0*
9. Quality of Service
    9.1 QoS Metrics: *throughput*, *end-to-end delay*
    9.2 QoS Level: N/A
    9.3 Cost: N/A
    9.4 Quality Level: N/A

Figure 4.3 Continued

developer provides the values to the various attributes of the UMM specifications after the QoS validation of the component (as discussed in section 2.1) has been carried out using the UniFrame QoS Framework [BRA02]. Like WSDL, these informal specifications are then refined into an XML format for further processing, like WSDL. An example of the XML representation of the UMM specifications has been shown in Figure 5.7 for the same component as in Figure 4.3. However, it should be noted that UniFrame does not bind itself to the use of XML and allows other standard formats for the specifications representation. In addition, since the UMM specifications of a component are in accordance to the knowledgebase for a pre-defined business reference model, it requires the component to advertise not only the manner in which it participates in the application but various other details as part of its service specifications. These details include features such as “Pre-Processing” and “Post-Processing” collaborators which define the interface specifications of the pre-processing and post-processing dependencies of the component respectively, with respect to the abstract component model specified as the Business Reference Model by the UniFrame’s knowledgebase. This in turn helps the in the automation of the composition of the DCS at the higher level. The UMM specifications also enhance the proposal of a multi-level contract for components proposed by Beugnard, Jezequel, Plouzeau, and Watkins (1999) [RAJ03]. They advocate that the contract of a component be made up of four levels: syntactic-level, behavioral-level, synchronization-level, and quality-level. The concept of a component specification in UMM goes beyond of what is proposed by Beugnard et al., in that it allows for the statement of various other details such as bookkeeping, collaborative, algorithmic and technological information, possible levels of service with

associated costs and effects of different environmental factors on the QoS parameters. The multi-level specification also aids in providing a better match during the discovery process.

Thus, while the WSDL document primarily indicates a web service's functions, such as input/output parameters and the transport protocols, the UMM specifications of a component play a more integral role in advertising the component and its composition in a distributed system.

4.1.3 Registries/Repositories

Web Services Registries: The Web Services framework supports two kinds of repositories - UDDI and WEB SERVICES Brokerages.

UDDI: The UDDI standardization provides for "searchable Web Services Registries" which facilitate the storage, discovery and exchange of information about businesses and their Web Services. UDDI is implemented in two forms:

UDDI Business Registry: publicly accessible and maintained by Microsoft, IBM, Hewlett Packard and SAP.

UDDI Private Registry: accessible only to authorized users.

The various entities involved during the utilization of UBR (UDDI Business Registry) [DIE03] are:

Operator Nodes: The organizations that host the implementation of the UDDI Business Registry are Microsoft, IBM, SAP, and Hewlett Packard. UBR operates on the principle of "register once and publish everywhere". This in turn implies a replication of the data within the operator nodes so that all instances of records are identical with each node. Operator nodes synchronize their information at least every twelve hours.

Custodian: The custodian for a company is the operator node with which it publishes its web services. A company can register and update its information only through its

custodian. This prevents multiple versions of the data from entering in the four different operator nodes.

Registrar: These organizations do not host implementations of the UDDI but act as assistants for organizations in creating data (such as business and service descriptions) and publishing in the UBR.

Structure and Information Model of UDDI: XML forms the basis of the overall information structure of UDDI which can be broadly divided into following information levels:

White Pages: General information about the provider, such as its name, contact information and identifiers.

Yellow Pages: Categorization of the providers' information based on their services.

Green Pages: Technical information about the provider's services or products. Usually contains references to the WSDL documents of the services enabling the client to know as to how to interact with the Web Service.

UDDI supports certain APIs for the clients to use the registry. These include:

Publishing API – It supports the publish operation on the UDDI Registry. The access to this API is restricted to authorized users only. Operator nodes implement a form of Authentication protocol to allow legal organizations to access this API. By means of publishing API, an organization is able to execute commands to create and update information in its operator node.

Inquiry API: Supports the find operation in three different patters (browse, drill-down and invocation). This API is accessible to any individual on the UBR who wishes to locate a service or a kind of service.

Web Service Brokerages: The Web Service brokerages are web sites that house information about the available Web Services in the form of a list, along with their web addresses. These brokerages can also supply additional services, which can include advanced search capabilities based on category, organization name or schema type,

service monitoring and service support, which can include services-related resources such as a tool that validates WSDL documents. Examples of some of the current Web Services Brokerages are: Allesta Web Service Agency, SalCentral Service, Xmethods and serviceFORGE.

UniFrame Registries: In the case of UniFrame, the entity that houses the information about components developed using a particular model is local to that component model. This entity is named "Active Registry", and is an enhanced version of the native registry of the corresponding object model. It has features such as:

Activeness – ability of the registry to listen to multicast messages from the headhunter and then establish communication, and

Introspection Capabilities – capabilities to introspect the registered components for their UMM specifications.

The conceptual difference that exists between registries of the two frameworks is in the way the registries participate in the discovery process of the components. In the case of the WEB SERVICES framework, the onus of locating components lies in the hands of the service requesters. While in UniFrame, the emphasis is on the automated discovery process provided by means of the URDS. Whether an organization needs to deploy one active registry per machine or one per many, is not decided and could vary depending on the size and necessity of the organization. While a service requester and publisher has to confirm to the underlying implementation of the UDDI registry as preferred by the company hosting it, the Active Registry is not as rigid and constraint since it builds upon the same native technology used for the development of components registered with it. The following table outlines the main differences in the registration mechanism of the two paradigms.

Table 4.3 Registration mechanism in Web Services and UniFrame paradigms

	REGISTRY		
Characteristics	Web Services – UDDI		UniFrame – Active Registry
	UDDI Business Registry	UDDI Private Registry	
Maintained By	Microsoft, Hewlett Packard, IBM and SAP	The Company hosting the web services, for its private use	The entity hosting the components to be registered in that Active Registry
Discovery	Indirect Discovery	Direct Discovery	Part of the UniFrame Resource Discovery Service Framework
Access	Public access (Both Individuals and organizations)	Only to Authorized members, decided by the company which hosts the registry	Organization hosting the components in that Active Registry
Contents	Locations of WSDL Documents	Locations of WSDL Documents	Reference to the registered components. The UMM specifications are periodically retrieved from the components to ensure the freshness of the specifications
Interface	Custom APIs of the UDDI specifications	Custom APIs of the UDDI specifications	Native to the component model in which the Active Registry

4.1.4 Quality of Service Assurances

Quality of Service Assurances in Web Services: Currently, service providers typically employ third parties to audit their Web Services during the creation stage as well as for reevaluation of the service on regular basis. An *auditor* achieves this in the form of testing and certification. Auditors may also be employed by the service requestors in order to gain a kind of guarantee about the level of service offered by the Web Service. The entire scenario employs "Service Level Agreements (SLA)" [DIE03]. These are "legal contracts in which a service provider outlines the level of service it guarantees for a specific Web Service". When customers purchase the Web Services subscription, they receive the services according to the quality-related contents specified by the SLAs. The service developer may maintain the SLAs. As the contents of the SLA are determined by the participating entities, there are no formal guidelines to specify the

level of service a particular Web Service provides. The QoS requirements, which SLAs of Web Services's outline, include a*vailability, accessibility, integrity, performance, reliability, conformance to standards and security.*

Quality of Service framework of UniFrame: Under the UniFrame paradigm, the components provide their QoS assurance to the interested clients by employing the UQoS framework. The UQoS framework provides a set of guidelines and the necessary platform that facilitates the publication, selection, measurement and validation of the QoS parameters of the components (as well the composed system). This is achieved by the framework with the help of three parts to its approach. These are: a) QoS catalog, b) specification and measurement of QoS and c) composition and decomposition models for QoS parameters [RAJ03]. The component developer is provided with the QoS catalog which helps him to select and specify the necessary QoS values to be included in the UMM specifications of the component. As introduced in Section 2.1, the catalog acts provides the necessary vocabulary related to various QoS parameters that are important and need to be considered by the developer while developing components specializing in a specific domain. It also provides the well-accepted measurement models for the parameters. Since every domain has its own constraints with respect to the QoS attributes, the catalog aims to act as a checklist for both the component developer and a client interested in identifying and validating its QoS attributes. The static QoS parameters of a component can be measured by the component developer using the measurement models enlisted in the QoS catalog. However, if the parameter is dynamic, i.e. its value changes with the environment of the system execution, measurement model alone cannot suffice. Its value needs to be determined by an empirical execution of the component. The UQoS framework uses the principles of event grammars [AUG00] for the measurement of dynamic QoS parameters with respect to different environment configurations and usage patterns. Unlike Web Services, UniFrame's QoS framework not only focuses on assurance of component QoS but also of the generated system. The UQOs comprises of the composition and decomposition models for the various QoS parameters discussed in the QoS catalog; in the form of a set of rules to predict the QoS

values of an integrated system from given specific values of the component and vice-versa respectively. UniFrame's iterative approach to system assembly from components meeting user's query specifications is based on constructive calculations of QoS metrics on representative set of test cases.

Quantifying the quality of service of the individual Commercial Off The Shelf (COTS) components, which compose to form an integrated system with a predictable quality, is one of the critical part of the UniFrame Approach. The features of the UQoS framework can thus be enlisted as:

- An existence of a QoS catalog containing detailed descriptions about QoS attributes, their classifications, their evaluation methodologies and the interrelationships with the other attributes.
- An integration of QoS at the individual component and distributed system levels.
- The validation and assurance of QoS, based on the concept of event grammars.
- An investigation of the effects of component composition on QoS; involving the estimation of the QoS of an ensemble of software components given the QoS of individual components.
- A QoS-centric iterative component-based software development process to ensure that the end product matches both the functional and QoS specifications.

Table 4.4 briefly tabulates the notion of QoS assurance of Web Services and components under the UniFrame paradigm.

Table 4.4 QoS assurance under Web Services and UniFrame paradigms

QoS ASSURANCE	
Web Services	UniFrame
▪ No specific formula guides the creation of the contracts (Service Level Agreements) ▪ Guarantees based on third party reliance and testing tools provided by them	▪ Each UMM Component can formally incorporate quantized QoS attributes ▪ QoS metrics to advertise a component's level of service ▪ Guarantees based on evaluation methodologies with mathematical foundation

The above outlined architecture-based comparison provided comprehensive details of the architecture-related aspects of the Web Services and UniFrame. It indicates the differences in the adoption of the methodologies for the different areas of achieving system integration based on identified metrics. The next section now enlists the characteristics which are associated with the overall system integration model of the two paradigms and will be inherent to any system developed using these frameworks.

4.2 Model-based Comparison

- As mentioned in Section 3.3, Web Services are all about XML and it being a text-based standard. This in turn implies delays involved in parsing it, which may prove vital in performance-critical applications. XML uses two sets of redundant tags to mark up every piece of information it represents. The tags are usually written to be humanly readable, which makes the actual tags a lot longer than they need to be. Also, one character in a Unicode document can be up to four bytes. Four bytes in some other proprietary binary format used by technologies such as DCOM or RMI can hold a lot more information than just one character. The ability to serialize the data over a connection, parse it quickly and efficiently is what plays a vital role in applications interacting over the network [WEB02]. UniFrame, on the other hand, leverages the components in a way so that they are a part of an application while remaining within their own object-model. This allows for more efficient ways of communication.

- HTTP is the preeminent protocol to transfer Web Service content and is allowed a free access through firewalls. HTTP, although used almost everywhere because of its reliability and ubiquity, is also not the most efficient transport protocol [WEB02]. Some of the disadvantages of using HTTP as the communication mechanism were discussed in Section 3.3. HTTP relies on a constant connection between the client and server when a request is made. This constant connection causes an overhead in cases when the data that needs to be transferred is quite small. However, in the Web

Service's universe, many transactions are essentially asynchronous. This in turn implies that the response of a web service request is not guaranteed. HTTP was not meant to deal with this kind of asynchronicity. It also relies on only one side initiating communication and the other side only responding to the request. This approach inhibits true peer-to-peer exchanges through Web services. A newer version of HTTP aims to fasten communications by making use of compression, but some of the previous issues still need to be pondered upon. Other protocols such as SMTP, over which Web Services can be implemented, still do not provide a major breakthrough in this respect. The UniFrame' approach to achieve communication between two components is through the use of glues and wrappers. These are generated at the time of system composition and are flexible to allow the incorporation of different communication protocols for component interactions. This is due to the fact that the glue-wrapper generation approach can be based on the notion of connectors (based on the connector model outlined in Chapter 3) and parameterization of the connectors can depend on the components to be connected. Depending on the type of communication involved between the components, for example, data streaming, event-based or message passing, appropriate communication mechanism is incorporated in the connector. Thus, as UniFrame does not attach itself to a specific protocol, it can avoid some of the drawbacks, discussed above, and related to the usage of HTTP.

- There is a growing need by today's IT industry for an information highway for both internal application integration and electronic partner integration. This divides the "integration" process into the two categories of EAI (Enterprise Application Integration) and B2B (Business To Business) integration (or BI solutions introduced in Chapter 3). While EAI software or middleware provides the infrastructure to rapidly connect and interface information between an organization's internal applications, B2B connectivity solutions' focus lies in leveraging a corporation's partnerships with suppliers and customers by integrating their applications and business processes with these partners.

Though these technologies share several fundamental architectural principles, they possess different requirements. Some of the ones which distinguish them have been tabulated below.

Table 4.5 EAI and B2B Solutions requirements [PIN01]

EAI Solutions	B2B Solutions
User and Transaction security	Document Security
Full Application Integration	Just enough partner integration (i.e. whatever the interacting partner can do)
Process Automation	Document Exchange automation
Real-time communications and message delivery	Communications and message delivery that fit the partners' capabilities
Solutions must be robust for reliability, scalability etc.	Solutions must be good enough to communicate with partners
Standardize and leverage object data across systems	Work with numerous data definitions and standards

The Web Services at this point are new and still not standardized in the sense that they do not explicitly support strategic considerations such as security, transaction handling, or session contexts (although standardization for some of these topics are in progress as a part of the Global Web Services Architecture). These requirements usually prove to be of utmost importance in EAI environments. The loose coupling and disconnected nature of Web Services guarantees that the request and response scenarios will be somewhat unreliable, and the ability to hold a session or transaction context over a long period of time is unpredictable [WEB02]. ebXML (as discussed in Section 4.1.1) uses a top-down approach to produce a solution that more appropriately addresses large-scale business-to-business (B2B) scenarios. Web Services have general purpose architecture with inherent interoperability that supports B2B scenarios but UniFrame extends beyond in scope and functionality.

The above discussion indicates that though Web Services do not seem a good candidate for EAI solutions but an effective and flexible B2B solution. Organizations globally are becoming aware of the importance and need of integration across

disparate platforms. An organization with numerous applications needs EAI solution and corporations that are extending their processes with partners need B2B. The future holds potential for a solution set that provides the functionality for both the requirements frameworks. The UniFrame with its unbiased approach towards any of them is an attempt in this direction.

- Although UDDI registries, both public and private, offer a great deal of advantage in terms of an application integration of the participating companies, they have their own set of limitations too. Firstly, because UDDI is fairly new, it has not reached standardization in a complete way, which holds true for UniFrame as well. Secondly, the UDDI Business Registry poses the question of data reliability. UniFrame does not involve the notion of publicly accessible registries. The Active Registries only allow authorized entities to publish components and interacts with the headhunter, thereby reducing the threats of data compromise. The discovery mechanism of the UMM Framework involves the headhunter storing the data about the components after it retrieves it from the Active Registries. The duration of the time interval after which this process repeats itself can be controlled so as to guarantee the freshness of the data within the meta-repository of the headhunters. UDDI registries, although describe web services, do not evaluate them. It does not house the Quality-of-Service information about a web service and requires an extensive search on the service-consumers part to do so. UniFrame on he other hand, provides an extensive Quality-of-Service framework to do so.

The above discussion indicates that requirements for an integration platform vary over a wide range of characteristics and require the addressing of numerous challenges such as service descriptions, communication mechanisms, QoS service assurance, service registration and discovery, etc. Web Services and UniFrame are both attempts in this direction. The differences indicated outlined clear certain ambiguities that might persist in the perception of the two models providing a better relevance to DCS requirements.

The next section now investigates as to the possibilities of collaboration between the two models for a more comprehensive DCS solution.

4.3 Web Services and UniFrame Collaboration

As outlined above, the WS and UniFrame differ in their approaches and associated implementation techniques. However, they can complement each other to provide solutions for future distributed systems. UniFrame uses the GDM [CZA00] to describe the properties of domain-specific components and to elicit rules for assembling heterogeneous components. One possible approach to integrate WS in UniFrame could be to use WS as a mechanism to wrap heterogeneous components. Due to the open nature of WS, such an approach will ease the task of assembling heterogeneous components adhering to existing and new object models. Furthermore, since WS are weak in representing the business semantics of application domains, this will also lead to the enrichment of WS technology in terms of semantic representation by following a model driven approach for specific domain-specific component models. UniFrame can then automatically generate WSDL from the models with the help of generators. This is an area which needs further exploration and investigation and is out of the scope of the current context.

The chapter identified certain metrics of comparison between the Web Services and the UniFrame paradigms. The discussions also surfaced many of the differences that exist in between the two approaches in composing distributed systems. One of the identified metrics was the process of discovery and registration as carried out in the two models. This has created a foundation that leads towards the next goal of adapting the UniFrame discovery system to the .NET component model.

5 .NET-BASED UNIFRAME RESOURCE DISCOVERY SERVICE

As mentioned in the introductory chapter, the second objective of the thesis is to study the adaptation of the UniFrame's approach into the .NET component model. This chapter focuses on this objective. One of the experiences gained from the analysis performed in Chapter 4 was the identification of the different areas of the UniFrame model which can help assess the functionality of the framework when adapted into a specific component model. The URDS architecture belongs to the category of such identified areas; others being registration mechanisms, description of components and their QoS validation, etc. UniFrame's approach for creating a meta-model to compose heterogeneous distributed computing systems with QoS constraints necessitates the need for a scalable and automated discovery service which is capable of registering and locating heterogeneous software components belonging to a wide range of component models and which can be discovered on the basis of their service specifications and criteria. This requirement led to model of the URDS and also makes it a significant area to explore in terms of variation and heterogeneity within the architecture. The URDS was briefly introduced in Chapter 4 along with some of its main constituents, namely the ICB, Headhunters and the Active Registries. The aim of this chapter is to discuss the adaptation of the platform-independent model of URDS into the platform-specific model of .NET. The chapter outlines the issues that were faced during the experience. In addition, the performance of the adaptation is also analyzed with respect to another platform-specific realization of the architecture created using Java RMI [MYS04]. The creation of a platform-specific .NET URDS architecture, helped to analyze both the performance and the functionality of the URDS architecture and at the same time helped to assess the adaptability of the URDS architecture in more than one component models.

The following sections provide a detail description of the URDS and its experimentation with the .NET Computing Model.

5.1 General Architecture

The Architecture of the URDS, as proposed in [SIR01], is organized as a federated hierarchy and is depicted in Figure 5.1. The figure shows that at the topmost level is the Internet Component Broker (ICB). The ICB acts as an all-pervasive component broker in an interconnected environment. It encompasses the communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between heterogeneous components. The ICB consists of a collection of services comprising of:

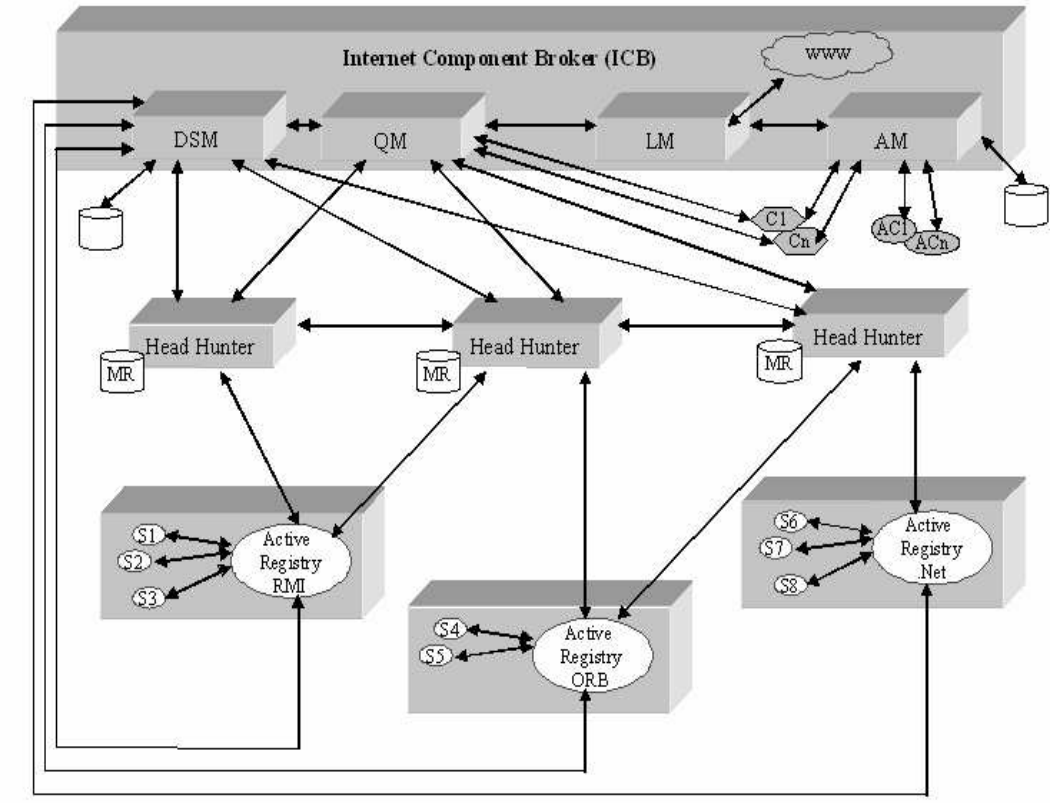


Figure 5.1 The URDS Architecture [SIR01]

Query Manager (QM): The purpose of the QM is to translate a system integrator's natural language-like query into the structured query language statement and dispatch this query to a selected Headhunter, which returns the list of service provider components matching these search criteria expressed in the query. The criteria used for selecting a particular headhunter are based on many factors such as the domain of the query. Requests for service components belonging to a specific domain will be dispatched to Headhunters belonging to that domain. The QM, in conjunction with the Link Manager, is also responsible for propagating the queries to other linked ICBs.

Domain Security Manager (DSM): The DSM serves as an authorized third party that handles the secret key generation and distribution and enforces group memberships and access controls to multicast IP address resources (that it assigns to its principals, namely the headhunters and the active registries to enable communication) through authentication and use of access control lists (ACL). DSM has an associated repository (database) of valid users, passwords, multicast address resources and domains.

Adapter Manager (AM): The AM serves as a registry/lookup service for clients seeking adapter components. The adapter components register with the AM and while doing so they indicate their specialization, i.e., which component models they can bridge efficiently. When the process of system composition entails components selected from different component models, there arises a need for the client (or the Glue-Wrapper Generator in this case) to contact the AM to search for adapter components that can match the needs of the client in terms of bridging different component models. This thesis also explores the issue of interoperability and provides a link to the concept of Adapter Components and Adapter Manager (in Chapter 6). However, a detailed discussion and implementation of adapter components, adapter manager and the associated connectors are not explored in this thesis.

Link Manager (LM): The LM serves to establish links with other ICBs for the purpose of federation and to propagate queries received from the QM to the linked ICBs,

if necessary. The LM is configured by an ICB administrator with the location information of LMs of other ICBs with which links are to be established. The LM is further discussed in detail in Chapter 6.

Headhunter (HH): The Headhunters perform the following tasks: a) detect the presence of service providers (exporters), b) register the functionality of these service providers, and c) return a list of service providers to the ICB that matches the requirements forwarded by the QM. The service discovery process performs the search based on multicasting.

Meta-Repository (MR): The Meta-Repository is associated with every Headhunter and serves as its local data store to hold the UniFrame specification information of exporters adhering to different models. The repository is implemented as a relational database.

S1…Sn: Services offered by components adhering to different component models (RMI, CORBA, etc.). These are identified by the service type name and the component's UniFrame specification which is stored as an XML specification outlining the computational, functional, co-operational and auxiliary attributes, along with zero or more QoS metrics for each component.

AC1…ACn: Adapter components, which serve as bridges between components implemented in different models.

C1…Cn: Component Assemblers, System Integrators, System Developers searching for services matching certain functional and non-functional requirements.

5.2 The .NET URDS-specific Architecture

The .NET-based URDS is a realization of the architecture discussed in the previous section with services being implemented using the .NET paradigm. A similar realization of the URDS using Java RMI paradigm is also implemented and experimented with [MYS04]. The aim of this .NET-based implementation is to highlight the technological differences that arise when the same service architecture is implemented under two different technologies. The main features of the RMI based URDS can be found in [SIR01]. This section describes the .NET URDS, specifically emphasizing the modifications needed because of the .NET paradigm.

The .NET URDS architecture uses .NET Remoting framework. This framework enables the development of distributed applications and is analogous to the Java-RMI framework. This factor is one of the reasons that led to the choice of the framework for implementation of the discovery service. The other factors, which contributed to choosing Remoting over Web Services, are as follows:

- Web Services can be built using any technology on any platform. Web Services provide a standards based and open communication medium while .NET Remoting is more proprietary to .NET. Since the focus of this exploration is to deal with principles unique to .NET and also to keep the option open to future extensions to incorporate Web Services, the implementation has been carried out with the Remoting framework.
- Generally, the term ".NET Web Services" implies "ASP.NET Web Services". However, in a distributed scenario, where Remoting enables the tuning of different features of the .NET distributed paradigm (such as the use of faster binary format over XML for the communication) to suit an application's requirements, it can provide with more performance and capabilities than ASP.NET Web Services.

Hence, all the entities of .NET URDS are implemented using the Remoting platform. Figure 5.2 shows the interactions between the main entities of the .NET URDS

[FRE02]. The figure shows the manner in which the communication between the three different entities - DSM, HH and AR, of the .NET-based URDS is carried out. The basic functionalities of each of these entities are what is defined by the URDS architecture and discussed in the previous section. The figure highlights the .NET specific communication mechanisms adopted and have been outlined in the following steps:

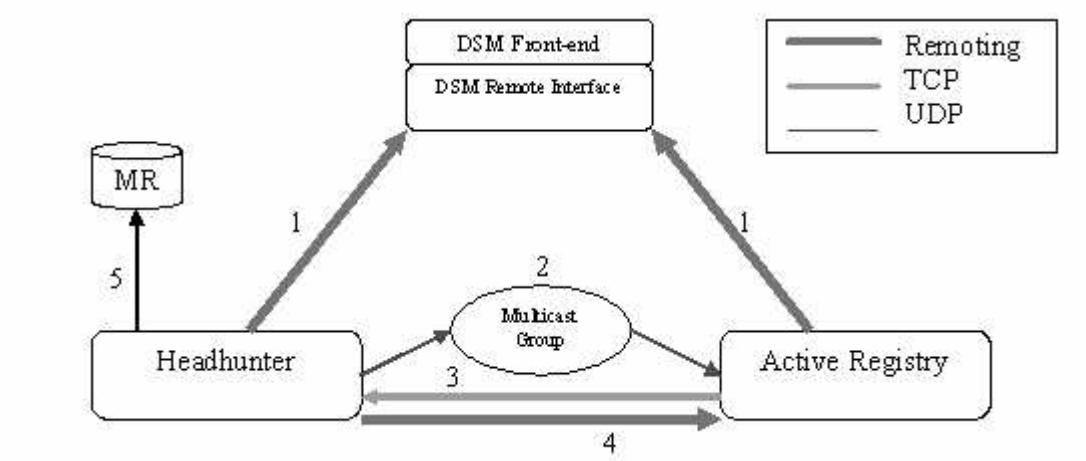


Figure 5.2 Communication between HH, AR and DSM in .NET URDS [FRE02]

1. The DSM Front end is hosted as a .NET application and hosts the remote server, DSM Remote Interface. The Headhunter and the Active Registry employ Remoting to communicate with this remote object. The communication can be carried out using either the HTTP or the TCP channels. The DSM remote server encapsulates the functionality of authentication and authorization. It also assigns the multicast group address to the HH and the ARs which register with it based on the domain in which they register.
2. Headhunter multicasts its encrypted location [SIR01] to the group address assigned to it and the Active registry, registered in the same domain as the Headhunter, listens at the group address. This communication has been implemented using the User Datagram Protocol (UDP) instead of using TCP or HTTP. The HH group multicasts at a periodic interval which can be as small as a few milliseconds and since the messages being sent from the HH to the AR are small enough to be encapsulated in a single datagram, UDP was chosen to be the underlying protocol.

3. After the AR receives the multicast message from the HH, the AR decrypts the message using the secret key (supplied by the DSM which controls all the security related issues of the URDS). It then contacts the HH using the location retrieved and unicasts its own encrypted location to the HH using TCP.
4. Once the HH receives the AR contact information, it then uses Remoting to establish the reference for further communication. The HH now periodically queries the AR for all the component information registered with it.
5. The UMM specifications retrieved by the HH from all the ARs it communicates with, are now stored in the local MR of the HH.

Further details on the above outlined implementation can be found in [FRE02]. The learning experience gained during this adaptation process is described below as one of the two main issues that were encountered during the .NET-specific adaptation of the URDS.

5.3 Issues in .NET Specific Adaptation

The implementation of the URDS architecture using the .NET Remoting Component model, revealed certain technological issues, which needed to be addressed in order to incorporate .NET component model within UniFrame. The following sections describe these issues.

5.3.1 Registration Mechanism

One of the critical issues faced during the development of the .NET URDS was the implementation of the Active Registry, because the .NET Remoting framework does not contain the notion of a registration mechanism that is analogous to the RMI registry mechanism. This observation was confirmed in [ING02]. It states that:

"*. . . unfortunately, there is no naming or registration service for .NET Remoting yet. therefore the URLs always have to be transferred "out of bound" (i.e. by means of email, copy-and-paste, or by developing your own naming service based on active directory for example). The client will therefore always need to know the complete URL to the server-side object.*"

The Remoting framework supports the notion of a registration class with which a .NET component registers its unique identifier using its APIs. This class is called the RemotingConfiguration Class. However, this registration mechanism is confined to the "Application Domain" of the application in which the component is hosted. The concept of Application Domain within the .NET Remoting context is briefly explained below [MIC01a].

Cross Application Communication - Application Domain: Typically, process boundaries are used to isolate applications running on the same computer. Each application is loaded into a separate process, which isolates the application from other applications running on the same computer. The applications are isolated because the memory addresses are process-relative; a memory pointer passed from one process to another cannot be used in any meaningful in the target process. In addition, direct calls between two processes are not allowed and require proxies for a level of indirection.

Managed code must be passed through a verification process before it can be run (unless the administrator has granted permission to skip the verification). The verification process determines whether the code can attempt to access invalid memory addresses or perform some other action that could cause the process in which it is running to fail to operate properly. Code that passes the verification test is said to be type-safe. The ability to verify code as type-safe enables the CLR to provide a level of isolation as the process boundary, at a much lower level of performance cost.

Application domains provide a more secure and versatile unit of processing that the CLR can use to provide isolation between applications. Several application domains can run in a single process with the same level of isolation that would exist in separate processes but without incurring the additional overhead of making cross-process calls or switching between processes. The ability to run multiple applications within a single process dramatically increases the server scalability.

The isolation provided by application domains has the following benefits:

- Faults in one application cannot affect other applications. Because, type-safe code cannot cause memory faults, using application domains ensures that code running in one domain cannot affect other applications in the same process.
- Permissions granted to code can be controlled by the application domain in which the code is running.
- Individual applications can be stopped without stopping the entire process. Application domains enable to unload the code running in a single application.
- Code running in one application cannot directly access the code or resources from another application. The CLR enforces the isolation by preventing direct calls between objects in different application domains. Objects that pass between domains are either copied or accessed by proxy. If the object is copied, the call to the object is local. That is, both the caller and the object being referenced are in the same application domain. If the object is accessed through a proxy, the call to the object is remote. In this case, the caller and the object being referenced are in different application domains. Cross-domain calls use the same remote call infrastructure as calls between two processes or between two machines. As such, the metadata for the object being referenced must be available to both the application domains to allow the method to be JIT-compiled properly. If the calling domain does not have the access to the metadata for the object being called, the compilation of the code fails. The mechanism for determining how objects can be accessed across domains is determined by the object.
- The behavior of code is scoped by the application in which it runs. In other words, the application domain provides configuration settings such as application version

policies, the location of any remote assemblies it accesses, and information about where to locate assemblies that are loaded into the domain.

Runtime Hosts - .NET: The CLR supports different kinds of applications such as Web server and Windows applications. Each type of application requires a runtime host to start it. The runtime host loads the CLR into a process, creates the application domains within the process, and loads user code into the application domains. The .NET Framework contains the different kinds of runtime hosts such as ASP.NET, Microsoft Internet Explorer and Shell Executables. Every .NET application requires a runtime host to execute.

Role played by Application Domains and Runtime hosts in the design of .NET AR: As has been stated earlier, UniFrame proposes that the AR registration mechanism for a component should be native to the component model and projects it as one of the characteristic feature of the UniFrame registration mechanism. The Java RMI-based URDS successfully provides this characteristic in its implementation [MYS04]. However, the role that the application domains and runtime hosts play in the design of the .NET AR, is significant in determining the adaptation of this characteristic in the context of .NET.

During the entire process, starting from a remote client getting a reference to the remote server to the client and the server establishing a communication for achieving the functionality of the application, the following three steps can be identified:

1. Register a server instance
2. Obtain Reference or Handle to remote server
3. Invoke functionality of the remote server

The process has been outlined for the Java RMI model in Figure 5.3 and is explained by the following paragraphs.

Java RMI Model's Remote Server Hosting Mechanism: Figure 5.3 shows the RMI registration mechanism native to the component model of Java RMI. The RMI remote objects register with a running instance of the RMI Registry at a well known location using the API of the RMI registry – Naming.rebind (URI). The object supplies its Unique

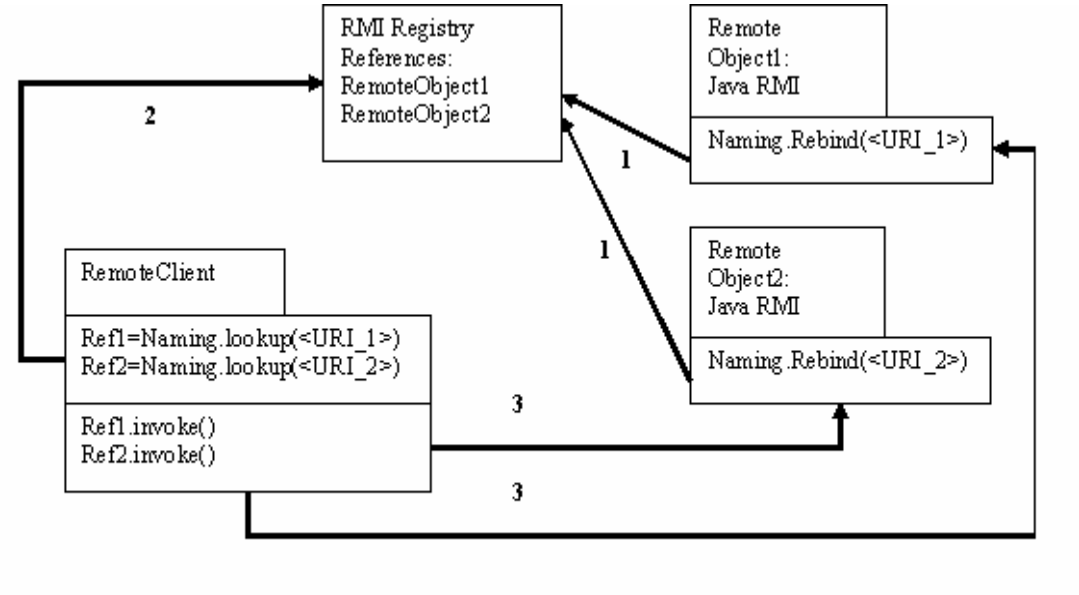


Figure 5.3 Registration mechanism in Java RMI component model

Resource Identifier (URI) in the process. The RMI registry stores a reference to these remote objects. A client can obtain a reference to these remote objects again using the API of the RMI registry – Naming.lookup(URI). The client supplies the URI of the remote object to obtain the reference.

Active Registry Adaptation in Java RMI Model: Figure 5.4 shows the registration mechanism of the URDS architecture adapted for the Java RMI model. It shows that the Java RMI Active Registry could enable the centralized registration mechanism for Java RMI components by wrapping the native RMI registry in a way such that now the remote object calls the AR.rebind(<URI>) with the same set of parameters as in the native mechanism. Since the AR wraps the RMI registry, the clients can still be able to call the Naming.lookup(<URI>) in order to get a reference to the remote objects. Hence, for the

RMI components, the AR was able to provide a registration mechanism which emulates the RMI's inherent registration mechanism.

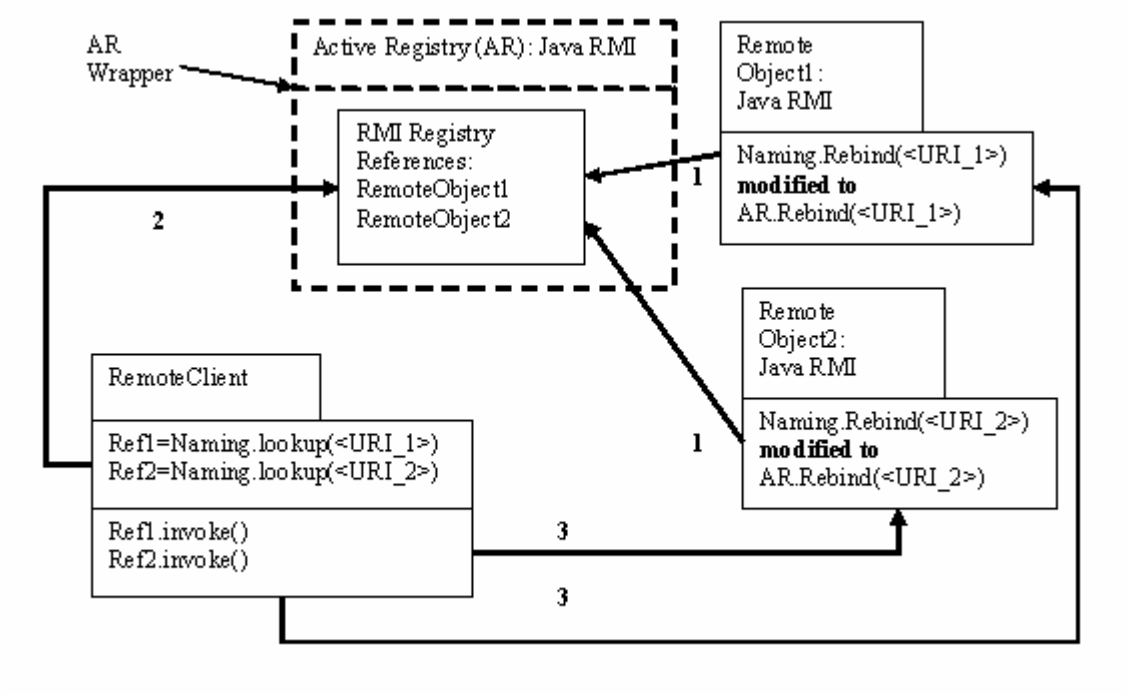


Figure 5.4 Active Registry-enabled registration mechanism in Java RMI

.NET Remoting Model's Remote Server Hosting Mechanism: The sequence of events which take place in order for a remote server to communicate with a client in case of the .NET Remoting paradigm has been depicted in Figure 5.5. The following are the observations from that figure:

1. Step 1, namely the process of registration, is missing in figure because of the lack of support in the .NET Remoting paradigm. The server utilizes the API of the RemotingConfiguration class in order to register the instances of remote objects with it and for the clients to get the reference to these registered remote objects. However, the scope of this class is bounded by the Application Domain in which the remote object is being hosted and the Remoting Configuration class of the framework does not exist as a registry (like the RMI registry). Its resources are

limited to the application domain of the application in which it is hosted. Hence, this class serves to only host a remote object. Under the UniFrame paradigm, every component is an independent entity and has its own set of resources and

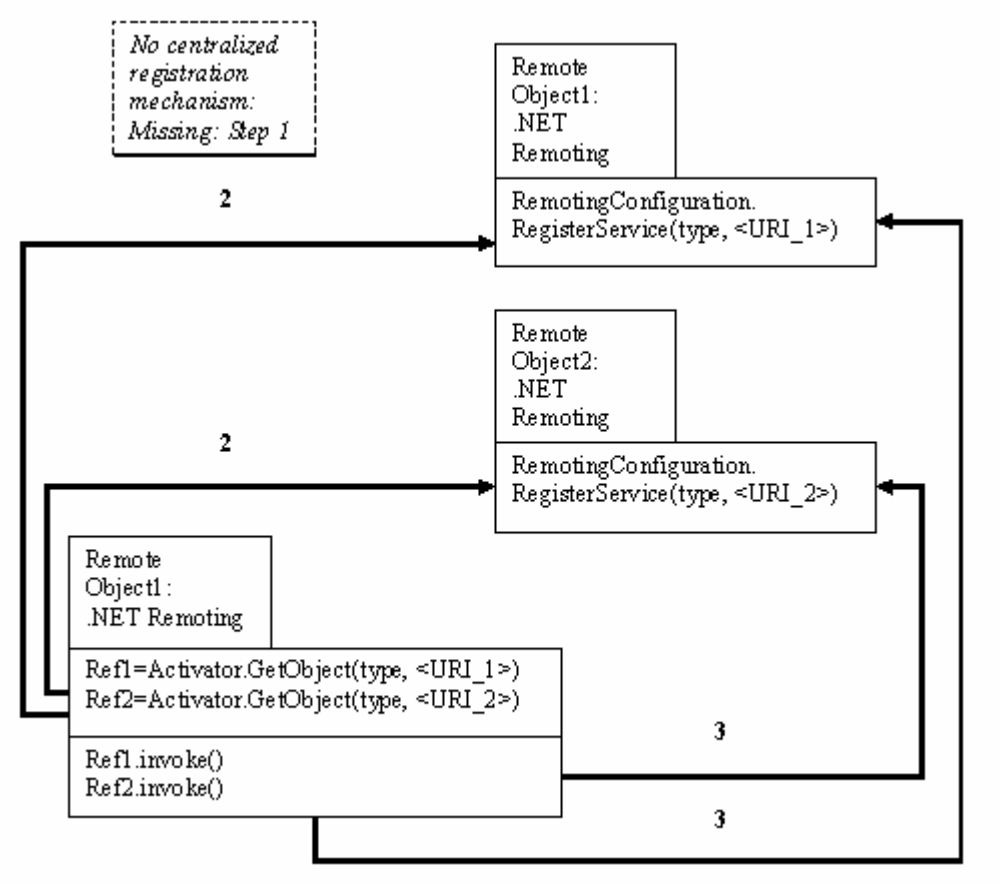


Figure 5.5 Registration mechanism in .NET Remoting model

dependencies. Thus, every time a .NET component is developed the runtime host for the application will load the CLR into a different process which will have its own application domain (as mentioned earlier). Hence, each independently developed component will have its own application domain and cannot be hosted in other application domain using the RemotingConfiguration, unless and until these is a customized registration mechanism to do so. The approach adopted as part of the study accomplishes this (to be discussed later).

2. The client knowing the URI (which will be the URL here) of the remote component (by means such as copy/paste or other explicit mechanisms) gets a reference to the remote object by contacting each remote object individually (since there is no single registry).
3. Client can now invoke the functionality of these remote servers.

Active Registry adaptation of the .NET Remoting registration mechanism (Figure 5.6): The above discussion indicates that for .NET Remoting there exists a need for some kind of central registration mechanism where components belonging to a particular category (or domain) can be registered and enable clients to discover them. In the context of the UniFrame paradigm, the solution was provided by the design and development of a .NET Active Registry. Hence, the Figure 5.5 was adapted to Figure 5.6 by the introduction of the .NET AR. It was created as a Windows Forms stand-alone .NET application that provides the features of an AR such as activeness and introspection capabilities. The registry is based on the principles of .NET Remoting and provides a registration mechanism for services to register and clients to query. This also places the missing link 1 where the components register with the .NET AR upon their startup using its API – AR.RegisterService(). The clients then knowing the URIs of the remote objects (by means of the discovery process of the URDS) can obtain a reference to these remote objects and finally in step 3, invoke their functionality.

The most notable aspect of the above adaptation is changing the mechanism for components to register their instances on the network. In the case of Java-RMI AR, though the name of the API now used by the component changed, the parameters supplied during the registration did not change (type and the URI). This is because the Java RMI AR is acting as a wrapper around the native registry of the RMI component model and continued to utilize its registration functionality. However, in the case of the .NET AR, since the AR needed to provide a registration mechanism from scratch, the information required by the AR from the component is more than just the type and the URI of the component. The .NET AR can be created in any of the application types or

runtime hosts. However, in each case, the CLR creates a separate *application domain* for its execution. In order for the AR to host the components within the same application domain, the .NET framework necessitated the AR to have access to the compiled assemblies of the components. The .NET AR then introspects the components to obtain their UniFrame specifications and return to the HH when queried for.

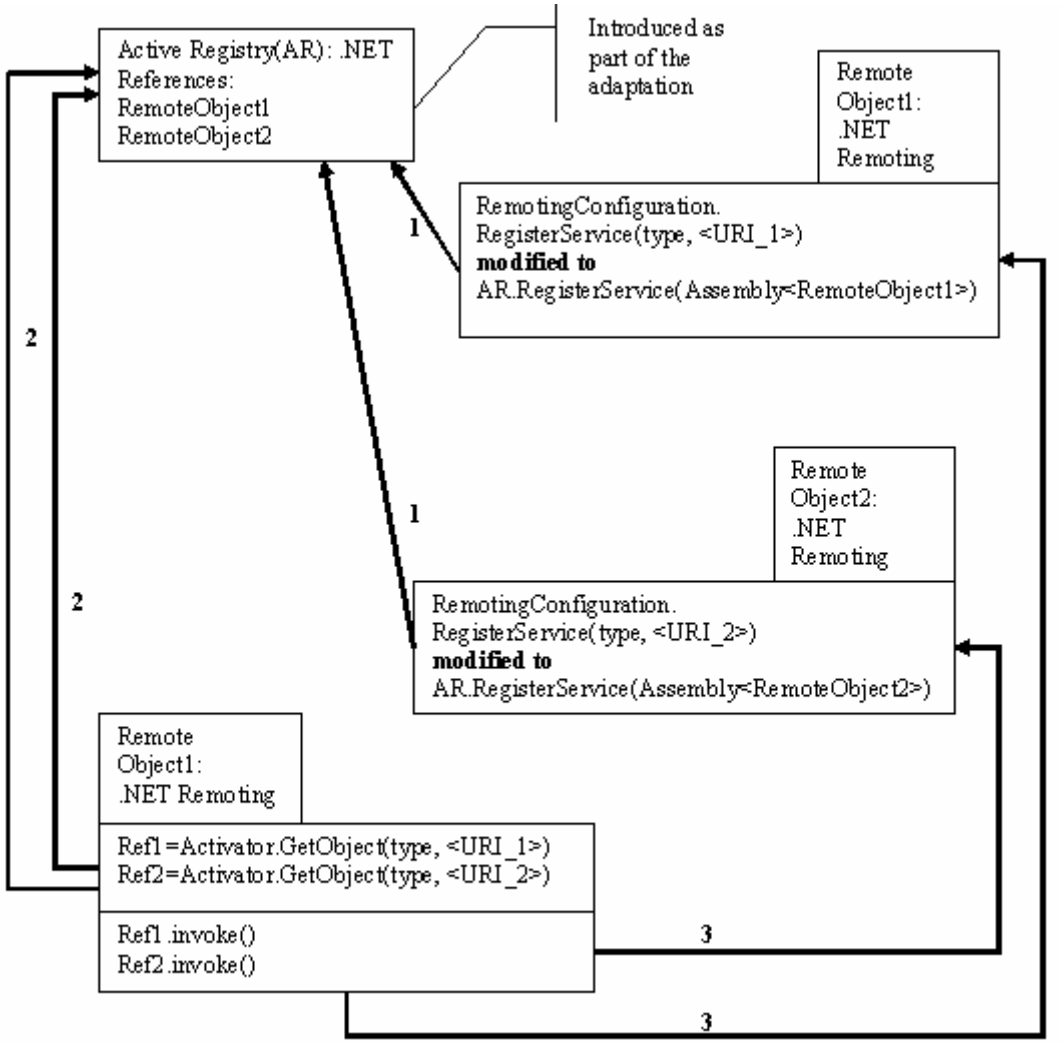


Figure 5.6 Active Registry adaptation in .NET Remoting model

The .NET Remoting allows for different modes in which components can be hosted, for example, the clients can be Client-Activated objects (CAO) or Server-Activated objects (SAO). The .NET AR is flexible and allows the component developer

to specify different parameters for hosting the components – such as the type of the channel (TCP/HTTP), object type (CAO/SAO) and port numbers.

5.3.2 Interoperability Issue

One of the other facets of the URDS architecture which was investigated in the context of .NET component model is the interoperability of discovery services adapted to distinct component models. Currently, these adaptations span across the Java RMI and .NET Remoting models. An experimentation has been carried out to compose a distributed system with the .NET and RMI instances of URDS running/available on the network which in turn facilitates an increased search space. While the following paragraphs focus on the interoperability issue experienced during the experimentation, Chapter 6 provides further details on it.

The systems involved in the integration can be broadly categorized into 3 categories – System Integrator, .NET URDS and the Java RMI URDS. The integration requires the flow of the query for components and the search results satisfying the query. The major challenge that is imposed during the experimentation is the propagation of this data across different component models. The following paragraphs provides the details of the issue.

As indicated in Section 2.1, the search process is initiated when a system integrator presents a query to the UniFrame. The general form of the query is a request to create a system that satisfies certain QoS parameters. For example, consider the following query expressed in a natural language like format:

"*Create an Account Management System that has availability >50% and end-to-end delay < 500ms*".

The name of the system is important in identifying the application domain of the system and the QoS parameters help identify desired properties of the system. The query is processed using the domain knowledge-bases that contains the key concepts from the

domain, a system architecture and the UMM descriptions of the components. From this query, a set of search parameters is generated which guides the component search process. The search parameters can be specified as the required UMM specifications of the components that should meet the criteria. Also, the specifications of the components that meet the search criteria can also be specified as the parameters of the UMM specifications template. Figure 5.7 shows a sample UMM specification template for a component "Cashier Validation Server" which forms a part of the Banking domain. Various aspects of the service can be specified as a part of the UMM-specifications. Nodes can be single-valued or multi-valued in which case they hold multiple child nodes. For example, the node, Domain name – Banking – is single-valued whereas the node - Auxiliary Attributes - is multi-valued since the auxiliary attributes of the component could be made up of various attributes such as security, mobility, etc. [HUA01]

```
<UMM_ConcreteComponent>
        <ComponentName> CashierValidationServer </ComponentName>
        <ComponentSubcase> CashierValidationServerCase2 </ComponentSubcase>
        <DomainName> Banking </DomainName>
        <SystemName> Bank </SystemName>
        <Description> Provide cashier validation service in banking. </Description>
        <ComputationalAttributes>
                <InherentAttributes>
                        <id> http://134.68.140.197:9000/CashierValidationServer</id>
                        <Version> 1.0 </Version>
                        <Author> Natasha Gupta</Author>
                        <Date> August 2002 </Date>
                        <Validity> Yes </Validity>
                        <Atomicity> Yes </Atomicity>
                        <Registration> http://134.68.140.143:8500/HeadHunter </Registration>
                        <Model> .NET </Model>
                </InherentAttributes>
                <FunctionalAttributes>
                        <Purpose> Act as validation server for cashiers in banking. </Purpose>
                        <Algorithms> <algorithm> JFC </algorithm>
                        </Algorithms>
                        <Complexity> O(1) </Complexity>
                        <SyntacticContract>
            <ProvidedInterfaces>
                        <Interface> IValidationCase2 </Interface>
            </ProvidedInterfaces>……..
              ……        <//PostprocessingCollaborators>
        </CooperationAttributes>
  <AuxiliaryAttributes>
 <Mobility> No </Mobility>
      <Security> L1 </Security>
          <FaultTolerance> L1 </FaultTolerance>
        </AuxiliaryAttributes>
        <QoS>
          <QoSMetrics>
    <Metric>
```

Figure 5.7 Abstract Component model [HUA01]

```
                    <ParameterName> throughput </ParameterName>
                    <FunctionName> validate </FunctionName>                    <Value> 9128.26 </Value>
                  </Metric>
                  <Metric>
                    <ParameterName> endToEndDelay </ParameterName>
                    <FunctionName> validate </FunctionName>
                    <Value> 109.55 </Value>
                  </Metric>
                </QoSMetrics>
                <QoSLevel> L1 </QoSLevel>
                <Cost> L1 </Cost>
                <QualityLevel> L1 </QualityLevel>
              </QoS>
</UMM_ConcreteComponent>
```

Figure 5.7 Continued

The example shown in the figure could be the UMM specifications of a component discovered based on the component query that was decomposed out of the system's query specified above. The component's query is also specified as parameters of the UMM specification template. For example, the component query could be composed of the following parameters:

Domain Name *like* "Banking"

Throughput >= 9000 ms

End To end Delay < 200ms

Availability > 50%

Based on the above search criteria, the component with the above UMM specifications matches the search criteria and hence, would qualify as one of the discovered components.

During the process of adaptation of the URDS to a specific component model, the UMM specifications – of the query and the description of the components discovered – were mapped to equivalent types, namely Query Component and the Concrete Component. Both the types are based on Abstract Component which defines the basic UMM specification template for a particular component and is stored in the knowledgebase. The Query Component defines the queried UMM specifications of a component and all the instances of the Concrete Component type entail the UMM specifications that actually matched the Query Component constraints for that component

type. Each of the types, whether Query Component or the Concrete Component, support a set of methods which enable the retrieval of data encapsulated inside the object.

Achieving interoperation between the two URDS instances required the propagation of these objects, namely query component and concrete components (for all the matching components), across the discovery services and between the system integrator. Since, the information to be handled could be large in terms of the number of components that satisfy the requirements of a single query, bridges were chosen to be the preferred interoperability mechanism due to the advantage they provide in terms of performance by allowing the serialization of data on wire to be in the binary format. A comparison of the different interoperability mechanisms was provided in Chapter 3. The above scenario was experimented with the use of the Ja.NET and iHUB bridges which were discussed in Chapter 2 as part of the related work. The experimentation reveals the following findings:

1. It is found in both the cases, Ja.NET and iHUB, that the bridges are limited in their capability to serialize objects across different component models. While an object's attributes could be successfully mapped into a serializable format (binary) and passed across the Java RMI and .NET, the interface of the object could not be mapped/serialized and then reconstructed on the other end of the bridge, i.e. in the context of the other component model. The experiment proved to be a test-bed for the Stryon Inc. [STR04] and led to a series of interactions with their technical team who acknowledged the limitation mentioned above and initiated certain steps to overcome it. The iHUB bridge was also found to be limited in its functionality to passing objects such as Arrays, ArrayLists and Hashtables, etc., which were the form of mapping of the multi-node attributes of the UniFrame specifications, shown in Figure 5.7.
2. One of the other findings gained out of the experiment was that the use of bridges in connecting heterogeneous components cannot be done in a completely seamless manner, i.e., requiring no code changes in the component. Some sort of customized code changes for each of the bridges were needed and re-compilation

was required to incorporate those changes within the components. This further implies that if the interoperation mechanism between any two components was to change, it would also imply change in the code of the component again. Based on the discussion of connectors in Chapter 3, the knowledge further provided a motivation to the use of connectors to mediate the interaction of heterogeneous components. The discussion was explained in detail as the problem of "Deployment Anomaly" in Section 3.4.1.

Solution Adopted: In order to address the issue of limited functionality of the bridges in serializing the interfaces of the objects across component models, the object in one component model was first converted to an intermediate format and this format was used during the marshaling/unmarshaling process. The intermediate format also uses the binary protocol of communication but encapsulates only the data values of the variables. The format is hence devoid of any interfaces and captures the state of the object being serialized. The result is an overhead involved in performing the translation of each of such objects transmitted across heterogeneous platforms.

The result of this finding was an attempt towards incorporating the use of connectors and further exploring the concept in the context of UniFrame further – for example the issue of automating the generation of the connector or glue/wrapper code in the context of UniFrame. The work done by [BUL00] is a possible approach and has been discussed further in reference to linking URDS instances in Chapter 6. The finding also indicates that any component which provides a possibility to connect to other components and participate in a collaboration with other components to form a distributed application, should provide a certain kind of interface which can be used to link the component to other components, homogeneous or heterogeneous, by means of a glue-wrapper code. There are a number of research initiatives in this direction and these are not discussed in this thesis. For the experimentation purpose, the interoperation was achieved by means of a distributed connector which was manually generated. The connector is depicted in Figure 5.8. The connector *frame* (namely the Provided and the Required roles, C1`Role

and C2`Role respectively), is tuned to the interface of the components to be connected (manually as of now and can be an identified metric requiring automation). Hence the C2 instead of invoking the interface of component C1, C1Role, now invokes the same interface of the connector, C1`Role.

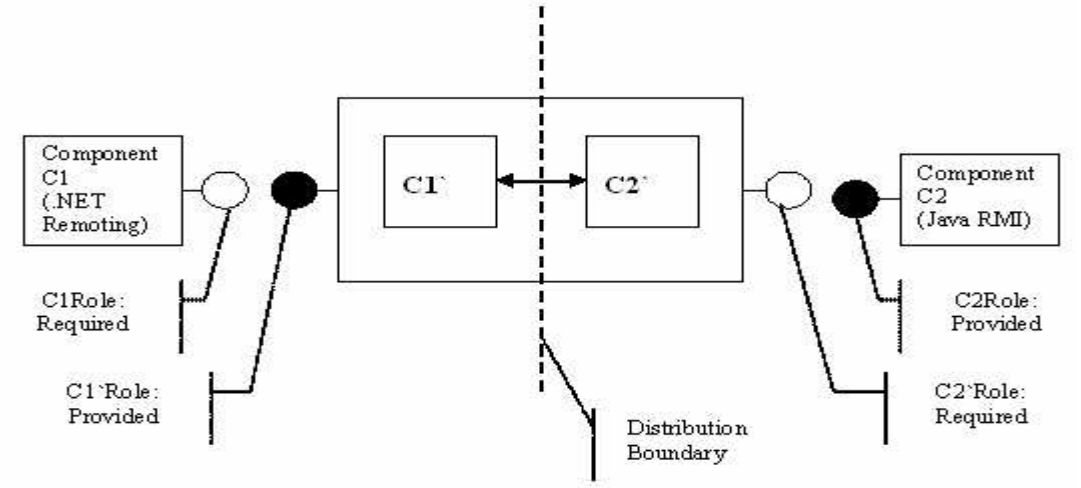


Figure 5.8 Connector mediation between .NET and Java RMI interoperation

The connector [BUL00] *architecture* is made up of the following primitive elements: adapter, stub and the skeleton, which provide the minimum functionality for the interconnection. There is no attempt on incorporating the QoS, interception or other such advanced features. The *adapter* performed the task of adapting the serialized objects into a format compatible with the interoperating mechanism incorporated within the other primitive elements of the connector. In this case, the primitive elements namely the *stub* and *skeleton*, incorporated the bridge, Ja.NET, and hence the adapter performed the task of translating the objects into a compatible format for the bridge as discussed earlier.

Having addressed the challenges in the .NET-adaptation of the URDS, it is necessary to validate the functionality of the solutions adopted. The thesis uses an empirical approach to validate by performing a set of experiments that are described in the next section.

5.4 Experimental Validation of .NET URDS

This section outlines the experimental validation of the .NET-based URDS undertaken for the purpose of the study. The experiments provided a means of testing the functionality of the adaptation achieved and are divided into the following two main categories:

a) *Comparison-based experiments:* Since the adaptation of a specific architecture, URDS, into a different component model has its own set of associated idiosyncrasies, the thesis adopts an approach of a comparative analysis between the Java-RMI and .NET-based adaptations. The analysis consisted of varying the number of one particular entity of URDS (such as HHs) and its effect was studied on the behavior of the URDS in terms of retrieving results for a given query. The behavior was studied for both the component models and their trends were compared. This gives an insight into the functionality of the URDS for the same variation but different component models.

b) *Experiments for performing check on the system's scalability:* After performing a comparison between the two component models based URDSs, the size of the .NET URDS system was increased to confirm its functionality on a larger scale.

Experimental Set-up: The experimental consisted of desktop Personal Computers with Windows 2000 operating system on a local area network. Each of the machines had the Microsoft .NET 2003 installed and hence, all the entities of the URDS and the components belonged to the .NET component model and were built using the same version of .NET.

Experimental Parameters: Since the entry point to the URDS system is the client who submits a query for the search of components meeting that query, the Client's Query Results Retrieval Time (CQRRT) has been taken as the parameter of functionality measure in each of the experiments. In addition, there are other variables associated with each of the experiments. These are:

N_H = Number of Headhunters

N_{AR} = Number of Active Registries

N_{QM} = Number of Query Managers

N_C = Number of Components matching the search criteria

N_{QC} = Number of the Querying Clients

N_{DSM} = Number of Domain Security Manager

Experimental Use-Case: The basic process that is inherent to every experiment is as follows:

1. The DSM is the first entity which is started and marks the beginning of building up of a URDS instance.
2. A certain number of Headhunters, Active Registries and components are deployed for the setup as required by the experiment. Every principal (HH and AR) of the URDS authenticate themselves with the DSM on startup.
3. A client submits a query to the QM.
4. The QM gets a list of available HHs from the DSM for the domain for which the query is targeted for. The QM then picks up a random HH, known as the Primary Headhunter (PH) and passes the query to the PH with the list of the other HHs in the list.
5. Every HH now employs a search algorithm [MYS04] to propagate the query further – the PH then combines the results obtained from each of the HHs with the results obtained from its own MR.
6. The results are returned back to the QM which returns the results to the Query Client.
7. The CQRRT is calculated from the instant the query is submitted to the QM to the time when the client receives the matching results (UMM specifications of the components matching the query) back. Depending on the experiment being performed, the number of entities in the system can vary and each time the CQRRT is measured to observe the effect.

Section 5.4.1 illustrates the first category of experiments, namely the comparison-based experiments and Section 5.4.2 outlines the ones related to scalability.

5.4.1 Comparison-Based Experiments

For the comparison-based experiments, there were a set of three experiments that have been identified in terms of the entities whose variation could be utilized in studying the behavior of the URDS system. These entities are the number of components (N_C), number of HHs (N_{HH}), and the number of Active Registries (N_{AR}). The following sections indicate these experiments in the order. The variation of number of queries and QMs has been studied as part of Section 5.4.2.

5.4.1.1 Experiment 1: Varying the Number of Components, N_C.

Figure 5.9 shows the trend that the CQRRT follows with an increase in the number of components that the URDS collects matching the criteria specified in the query. The rest of the parameters such as N_H, N_{AR}, N_{DSM}, N_{QM}, and the N_{QC} were kept constant and only the N_C was varied from a range of 1 to 24. The values of these parameters can be found in Figure 5.9. Figure 5.9 (a) depicts the trend that was observed in the case of a .NET URDS and Figure 5.9 (b) shows the observation for the Java RMI URDS. Each of the figures is accompanied by the number of other entities that existed in the system for the experiment. As can be seen, there is a linear variation in the CQRRT with an increase in the number of components – in both the cases. However, the .NET URDS shows a higher slope (600.32) than the corresponding experimental result of 113.7. There could be multiple reasons for this difference. The .NET HHs were all running on a different Windows machine with a completely isolated file system. However, the Java RMI HHs all shared the same file system on a UNIX-based server. This factor can add to the increase in the time it takes for the .NET PH to collect results from all the other HHs. One of the other possible reasons which could play a major role in the difference in the values is the amount of time it takes for the HHs to retrieve the component's UMM

.NET URDS performance: $N_{DSM} = 1$ $N_H = 1$; $N_{QM} = 1$; $N_{AR} : N_C = 1:4$; N_C = variable

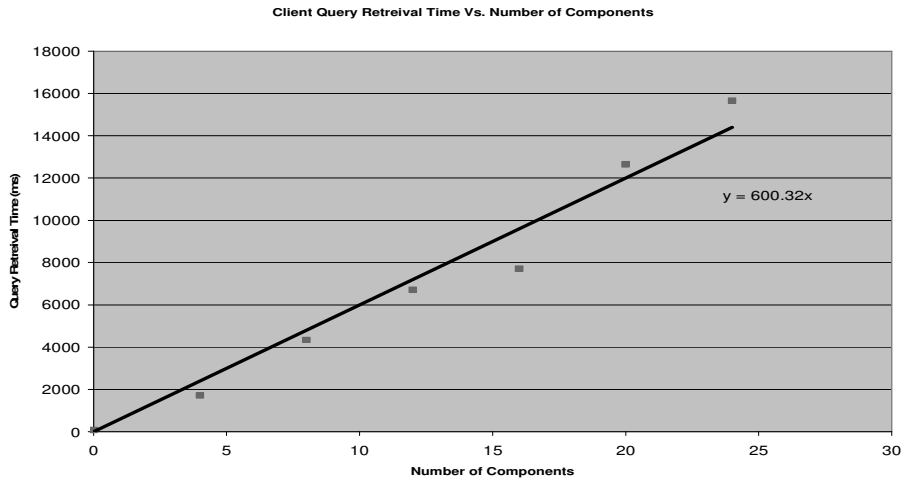


Figure 5.9 (a) Variation of CQRRT V Number of Components, .NET URDS

Java RMI URDS performance: : $N_{DSM} = 1$ $N_H = 1$; $N_{QM} = 1$; $N_{AR} : N_C = 1:4$; N_C = variable

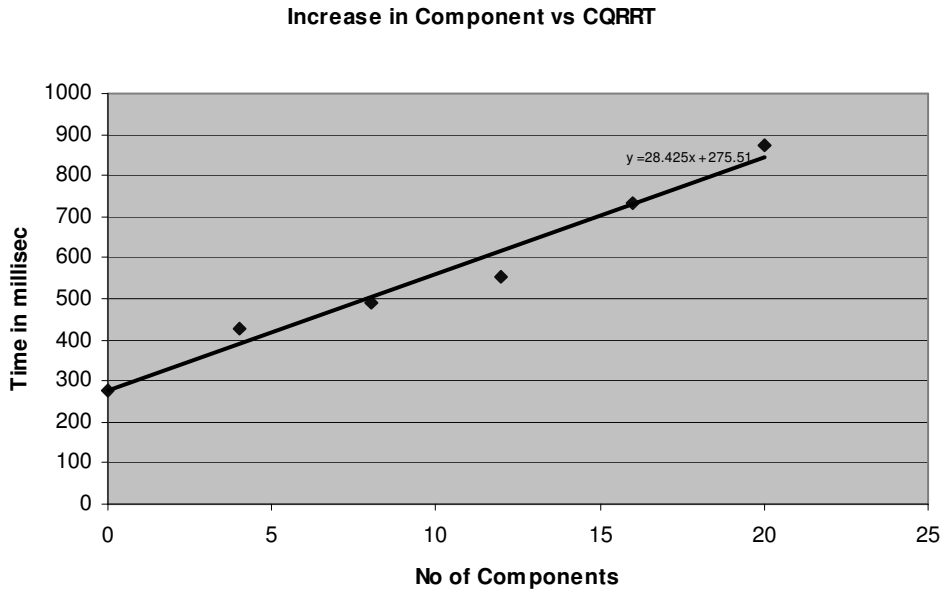


Figure 5.9 (b) Variation of CQRRT V Number of Components, Java RMI URDS [MYS04]

Figure 5.9 Increase in the Number of Components matching the criteria of the query V CQRRT

specifications from its own meta-repository. In case of the Java RMI URDS, the MR is implemented as a database using the Oracle version 8.0 database server. Whereas in the case of the .NET URDS, the MR is in the form of a MS Access database. Oracle server is associated with its own set of optimization techniques for the query execution. Whereas such an optimization is missing in the case of MS Access database which just serves as repository of information. Hence, as the number of components increases, it takes more time for a .NET HH to retrieve the information than a Java RMI HH.

Therefore, it can be concluded that the .NET URDS behaves in a similar manner as the Java RMI URDS, according to the trend followed by the graph, but there is a difference in the actual values for the time it takes for the client to retrieve results for its query. The difference is the outcome of the heterogeneous component models and associated environments to which the URDS has been adapted to.

5.4.1.2 Experiment 2: Varying the Number of Active Registries

A HH in the URDS periodically communicates with known ARs from its own domain. With every communication the HH updates its meta-repository with the set of results obtained from the AR. The process is periodic and hence, assures that the data store of every HH is up to date with latest information of the components registered with the ARs. Within the HH, the process of updating the meta-repository runs in the background. When the HH receives a query, the HH queries its meta-repository to search the matching components and returns it as a set of Concrete Components back. This process is independent of the number of ARs that might have contributed to the result set. Hence, an increase in the number of ARs does not affect the CQRRT may be except a few variations due to the processor usage. This trend is depicted in the graphs shown in the Figure 5.10. Figure 5.10 (a) shows that for the .NET URDS, the CQRRT remains almost constant with the increase in the number of ARs. In case of the Java RMI URDS, Figure 5.10 (b), the increase is negligible and hence follows a similar trend. More details about the Figure 5.10 (b) can be found in [MYS04].

.NET URDS performance: $N_{DSM} = 1$; $N_C = 1$; $N_H = 1$; $N_{QM} = 1$; $N_{QC} = 1$;

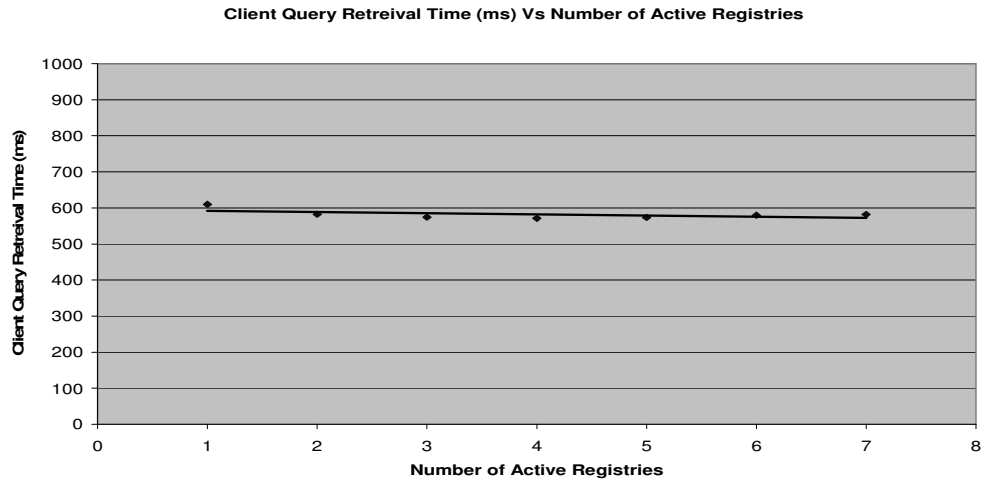


Figure 5.10 (a): Variation of CQRRT Vs Number of Active Registries, .NET URDS

Java RMI URDS performance: $N_{DSM} = 1$; $N_H = 1$; $N_{QM} = 1$; N_{AR} = variable; $N_C = 1$

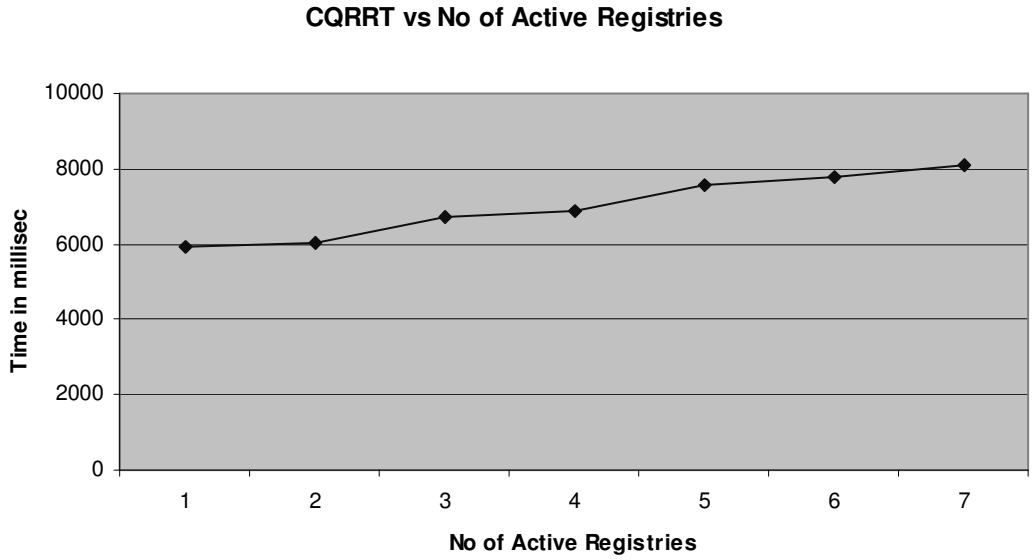


Figure 5.10 (b): Variation of CQRRT Vs Number of Active Registries, Java RMI URDS [MYS04]

Figure 5.10 Increase in Number of Active Registries V CQRRT

5.4.1.3 Experiment 3: Varying the Number of Headhunters

The experiment studies the behavior of the .NET-adapted URDS system with the increase in the number of the headhunters while keeping other parameters, N_C, N_{QM}, N_{AR}, and N_{DSM} as a constant factor. Specific values for these parameters have been outlined in Figure 5.11 (a). It can been seen from the figure that the graph has an increasing characteristic trend implying that as the number of headhunters spanning the search space increase, the longer it takes for the results to be retrieved by the client, i.e. the CQRRT also increases. The trend can be verified against a similar experiment that was done in the context of the Java RMI model with the same parameters. The graph has been depicted in Figure 5.11 (b). Both the graphs show a similar trend followed by the two systems for the variation in the number of headhunters. The trend can be attributed to the underlying propagation protocol that has been adopted for the query propagation. The protocol implements a simple query routing technique that is based on the random propagation of queries. The protocol will be briefly stated here and more details can be found in [MYS04]. When the QM submits the query to one of the headhunters as the PH, it chooses a particular "branching factor", b, to decide the number of HHs comprising its child nodes for the protocol. Every HH at the root of the tree implements the same value for b. The aim is to limit the depth of the propagation tree which by choosing the value of b. Every HH, receives a list of, say, k headhunters. The HH calculates k % b (k modulo b) = s = number of HHs that form the immediate children of this HH. Remaining HHs, r = k – s, are the HHs to be allocated by this HH. Now the HH calculates the number of other headhunters to be passed to each headhunter in the subset s. This number is p = r/s. Hence, for every headhunter in the next level, k = p. The same procedure is applied by every HH at its own level with its own set of HHs k. therefore, as the number of HHs in a given system increases, k increases at every level (keeping the b constant) and hence the depth of the tree also increase. This results in an increase in the CQRRT. The trend is confirmed by the .NET and Java RMI based URDSs. For a fixed value of k (for the PH), as the value of b increases, s increases (k % b = s) and hence r decreases. This implies a decrease in the depth of the search tree which implies a decrease in the CQRRT. This can imply a smaller slope for the trend line in graphs of Figure 5.11.

.NET URDS performance: $N_{DSM} = 1$; $N_C = 1$; NHH = variable; $N_{QM} = 1$; $N_{QMC} = 1$;

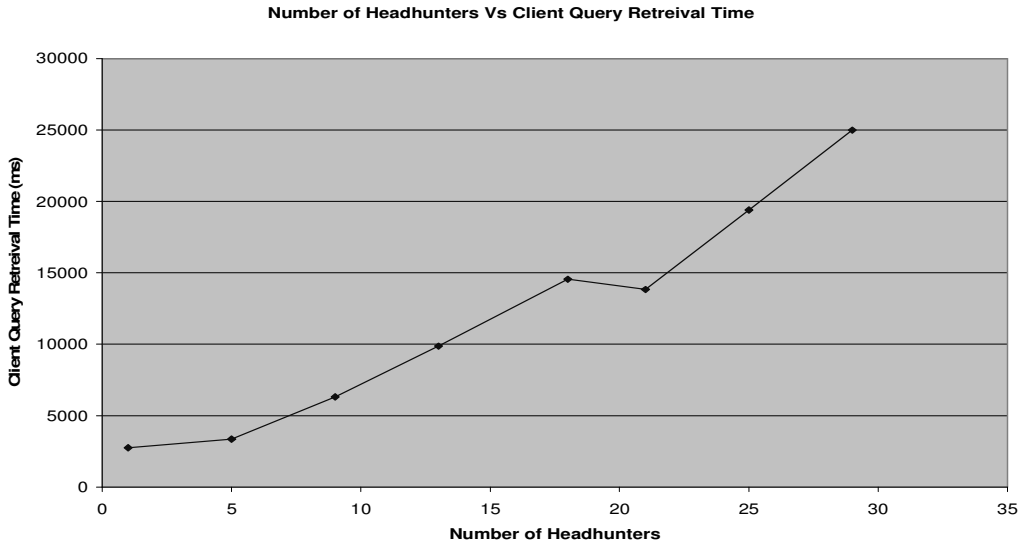


Figure 5.11 (a) Variation of CQRRT Vs Number of Headhunters, .NET URDS

Java RMI URDS performance: $N_{DSM} = 1$; $N_C = 1$; N_{HH} = variable; $N_{QM} = 1$; $N_{QMC} = 1$

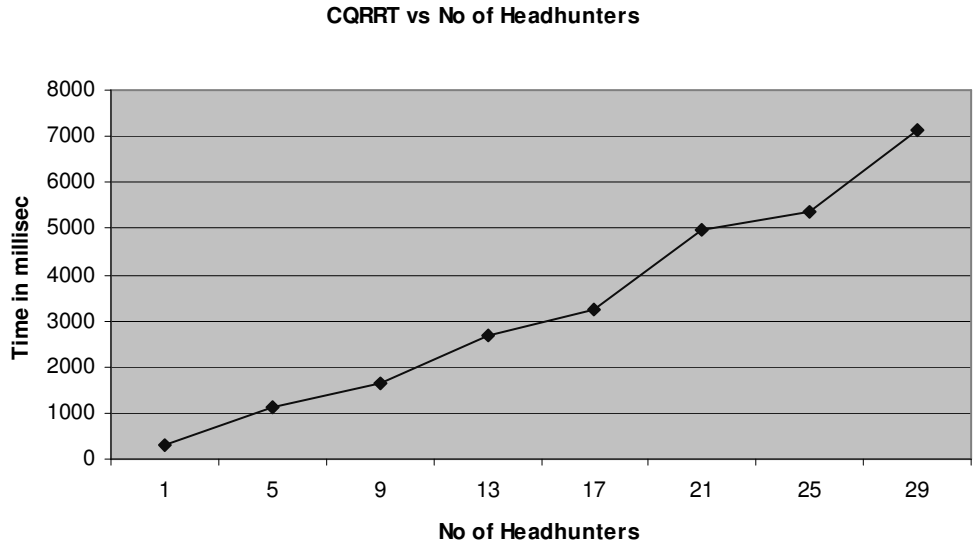


Figure 5.11 (b) Variation of CQRRT Vs Number of Headhunters, Java RMI URDS [MYS04]

Figure 5.11 Increase in Number of Headhunters V CQRRT

5.4.2 Scalability-Based Experiments to Check the System Functionality

5.4.2.1 Experiment 4: Varying the Number of Queries in the System

One of the experiments performed for assessing the .NET-specific URDS system's functionality at a bigger scale was in terms on studying the CQRRT variation with the number of the queries that exist in the URDS system during the same period of time. For this, the experimental setup consisted of a fixed number of HHs, ARs and DSM. There were also 2 QMs that were deployed to handle the incoming queries to the system. There are a number of query clients running and each client injects a query into the system through one of the QMs. Each client then calculates the time it takes for the it to obtain the results for its query. For a known value of the number of queries into the system n_i (i=10, 20, 30, ….), a certain number of clients were run kj (j=1, 2, 3, 4…) and the average response time for each value of i was calculated. As the number of queries increase, increasing i, the trend was observed and plotted in Figure 5.12. It can be observed that as a number of queries within the system increases, the CQRRT also increase with nearly a second order trend line. A similar experiment was also performed in Java RMI with N_{QM}=1 and rest of the parameters remaining the same. The Java RMI experimentation also shows an increase in the CQRRT as the number of query clients accessing the system increase. The results of the Java RMI-URDS experiment are shown in Figure 5.13. The difference in the trends could be attributed to many factors. Difference in the component models, difference in the number of entry points of the queries to the system and environment in which the experiments were conducted are some of the factors.

5.4.2.2 Experiment 5: Adaptability of the System at a Larger Scale

After studying the effects of varying the number of different parameters, an experiment was conducted in order to take a snapshot of the .NET URDS depicting it's adaptability to a large number of entities.

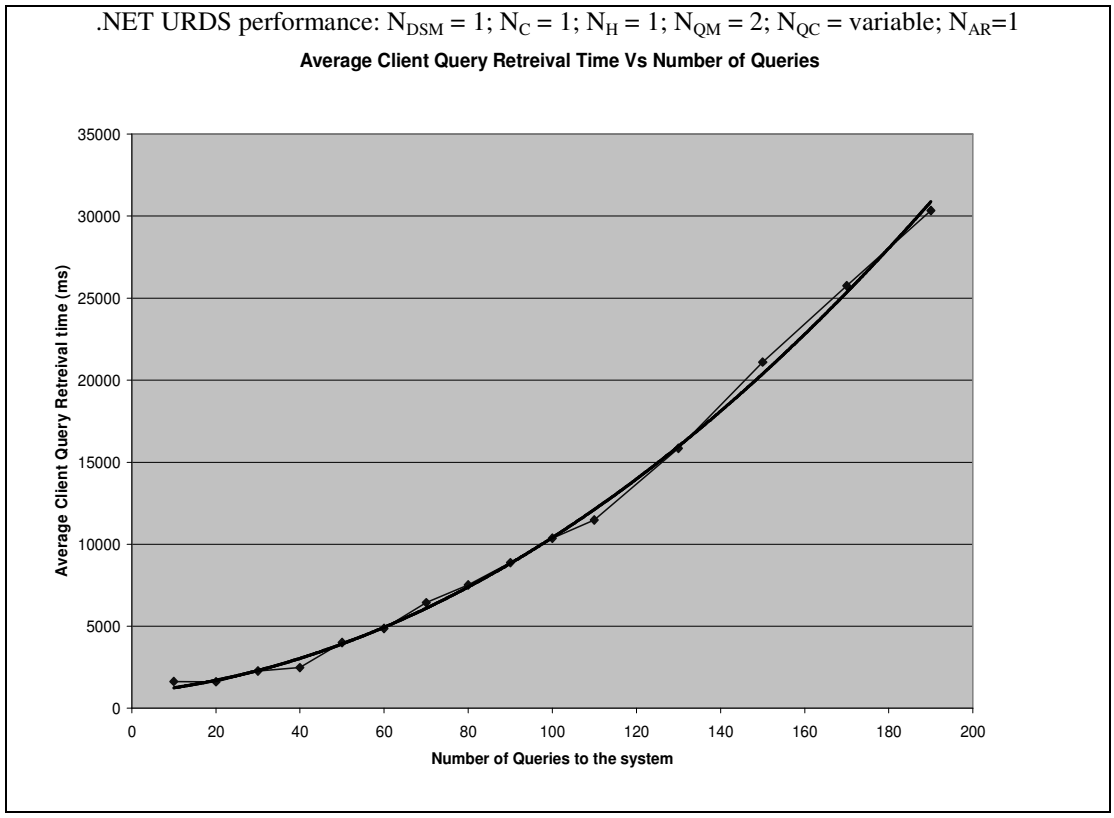


Figure 5.12 Increase in Number of Queries V CQRRT

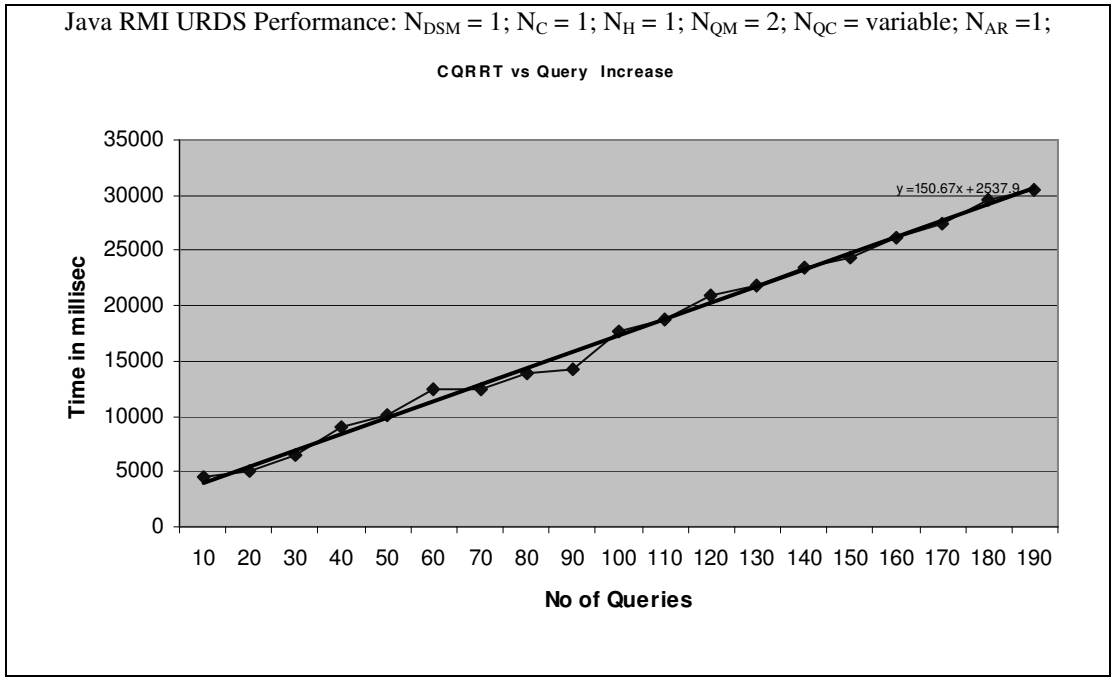


Figure 5.13 Variation of the Number of Queries, Java RMI URDS

When the QM is configured as a TCP server, it accepts .NET Remoting requests over TCP/IP. For the experimentation, the configuration settings for each QM were taken as the default provided by the .NET Remoting framework for each TCP port opened by the server. The number of clients and QM in the large-scale experimentation were chosen after trial and error experimentation with a larger number of clients and fewer QM. However, with the default configurations for the TCP server (QM in this case), the error - "request queue full" was thrown. This happens when number of incoming requests exceeds the size of the request queue, used to buffer accepted requests which have not yet been serviced. A larger initial request queue size may improve the server's ability to buffer large numbers of simultaneous connection requests, at the cost of additional memory use though. However, since the focus of the experiment conducted was to test the scalability of the URDS model without any modifications to the default configuration of the underlying object model, here .NET Remoting, the experimentation was performed with the default queue size, thread pool size etc. The parameters are as specified with the graph in Figure 5.14 and were chosen after a trial and error approach and arriving at the numbers in which the experiment successfully completed in about two and a half days. The following graph only shows the behavior of the four of the total nineteen clients (in terms of the CQRRT) which contacted the same QM for the retrieval of the results. It can be observed from the graph that the average response time of the clients vary in the range of 756.500s (12.75 minutes) to 906.5 ms (15.10minutes).

The scalability of the QM, as a TCP server, under the .NET Remoting paradigm can be further extended by setting up the following parameters of the TCP Server [BRI04]

TCP Backlog: TCP backlog (connection queue length) used by the server socket. Larger values may improve the system's ability to accept multiple simultaneous socket connection requests. If zero, the system default backlog is used.

Thread Pool – Minimum Size: Minimum number of threads in the thread pool. This is the number of threads started by the thread pool manager when a TCP server is started. The thread pool will never shrink below this size, regardless of thread use.

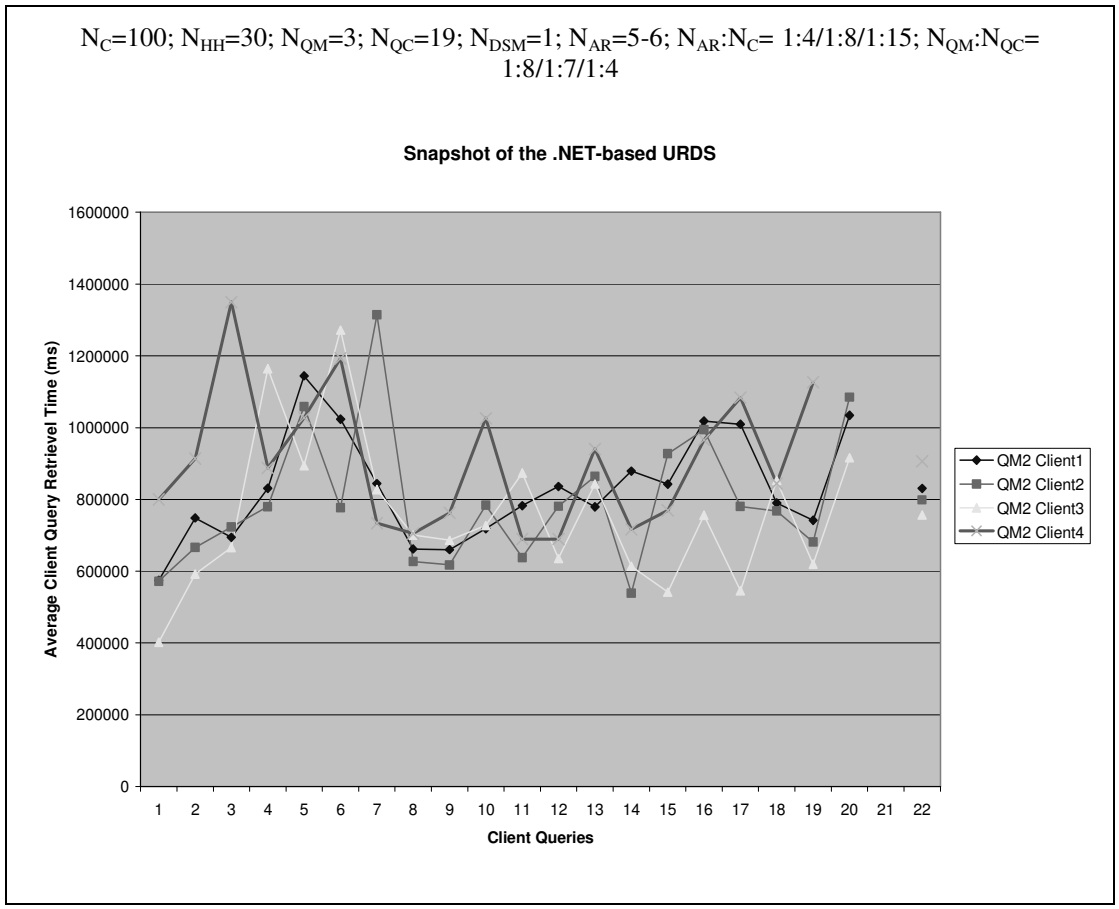


Figure 5.14 Scaled .NET URDS behavior (for only four clients)

Thread Pool – Maximum Size: Maximum number of threads in the thread pool. The thread pool will not grow beyond this size, regardless of thread use.

Thread Pool – Minimum Available: Minimum number of available threads before additional threads are started. If the current number of threads in the thread pool is less than the *Maximum size*, the thread pool will attempt to add more threads ("grow" the thread pool) to keep up with incoming requests. Typically set to 1.

Request Queue – Initial Size: Initial size of the request queue. The request queue is used to buffer accepted requests which have not yet been serviced. A larger initial request queue size may improve the server's ability to buffer large numbers of simultaneous connection requests, at the cost of additional memory use.

Request Queue – Maximum Size: Maximum size of the request queue. This parameter restricts the number of requests that can be buffered. Once the size of the request queue reaches this value, additional connection requests will be rejected. If zero, the queue grows as needed.

The study of the URDS architecture extended to the .NET component model revealed certain issues that need to be considered when incorporating heterogeneous component models within the context of UniFrame. Each of the features in this chapter is analyzed towards encompassing of the .NET component model into UniFrame. The chapter provides the guidelines for this, particularly into the registration and discovery mechanism of UniFrame's approach. The chapter addresses the issues for this incorporation and also provides an architecture for the Active Registry to tackle these issues. The architecture is validated for performance with two component models and hence provides an assessment platform for the adaptability of the URDS architecture. However, the issue of Heterogeneity and Interoperability, a major challenge for UniFrame, exists and needs to be discussed within the context of the .NET component model. Having discussed the .NET URDS architecture in this chapter, the next chapter addresses the issue of interoperability with respect to linking heterogeneous discovery services.

6 LINKING UNIFRAME RESOURCE DISCOVERY SERVICES

The third objective of the thesis has been undertaken in this chapter, namely to address the issue of heterogeneity within UniFrame. One of the studies outlined in Chapter 1 for the study of UniFrame in the context of .NET was to experiment and analyze the adaptation of the URDS with the perspective from the .NET component model. Chapter 5 provided for such as analysis with the architecture of the .NET discovery service and the issues faced during the process. The chapter also outlined a performance analysis that was carried out between the performances of the adaptation of the URDS into two different component models. The analysis consisted of studying the variation of different parameters of the discovery service and measuring them against the time period that elapses between the invocation of a query submission interface and retrieval of the results for the same. The study leads to another question – If there could be multiple instances of the discovery services on the network, can they be integrated in order to provide for a more scalable and comprehensive solution? What are the issues that need to be considered in such a scenario? How can the heterogeneity that exists between different discovery services be resolved? All these questions are discussed in this chapter.

Figure 6.1 [SIR01] depicts a hierarchical nature of the URDS architecture in order to achieve federation between UniFrame discovery services. This allows the expansion of the search space of the URDS and consists of the ICBs of different URDSs linked together to form a federation. The Link Manager (depicted as LM in the figure) performs the function of linking the ICBs. [SIR01] also discusses some of the basic algorithms that a LM should provide in order to achieve its functionality. However, it does not provide

the LM architecture in detail. This chapter focuses on the LM with an underlying aim to also analyze the issue of handling heterogeneity – placed within the context of discovery

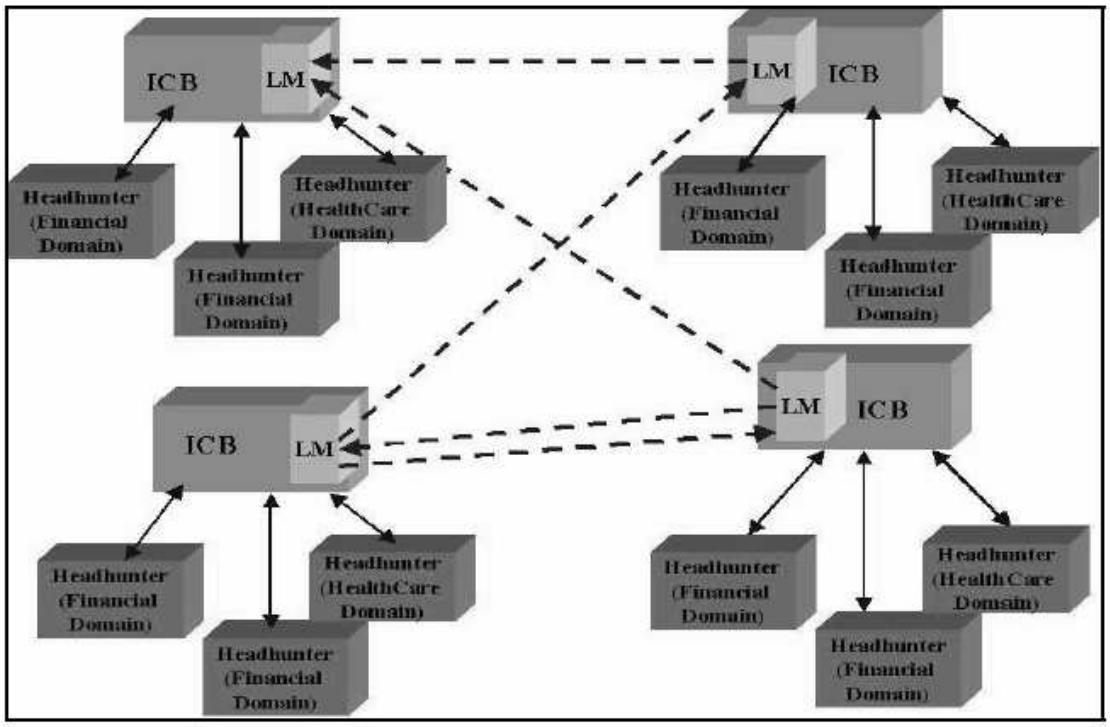


Figure 6.1 Federated hierarchical organization of ICBs [SIR01]

service, the suggested approach of the connectors discussed earlier, and .NET component model with the other existing URDS adaptation in Java RMI. However, federation requires more than just a LM in order to facilitate an efficient and extensible interconnection. The next section discusses such an architecture.

6.1 Proposed Architecture for Linking URDSs – Discovery Manager

Each URDS contains one LM dedicated for creating a federation of discovery services. If there are multiple instances of the URDS, there exists a need for a well-defined architecture that would govern the propagation of the query within these instances and the collection of the results from them. The proposed architecture is shown in the Figure 6.2. The figure introduces a “Discovery Manager” (DM) which is stands at

one level above the ICB in the hierarchy of the URDS. The DM is needed to mediate between the different URDS instances and the System Integrator (SI), introduced in [HUA03], which performs the function of composing the actual distributed application under the UniFrame paradigm.

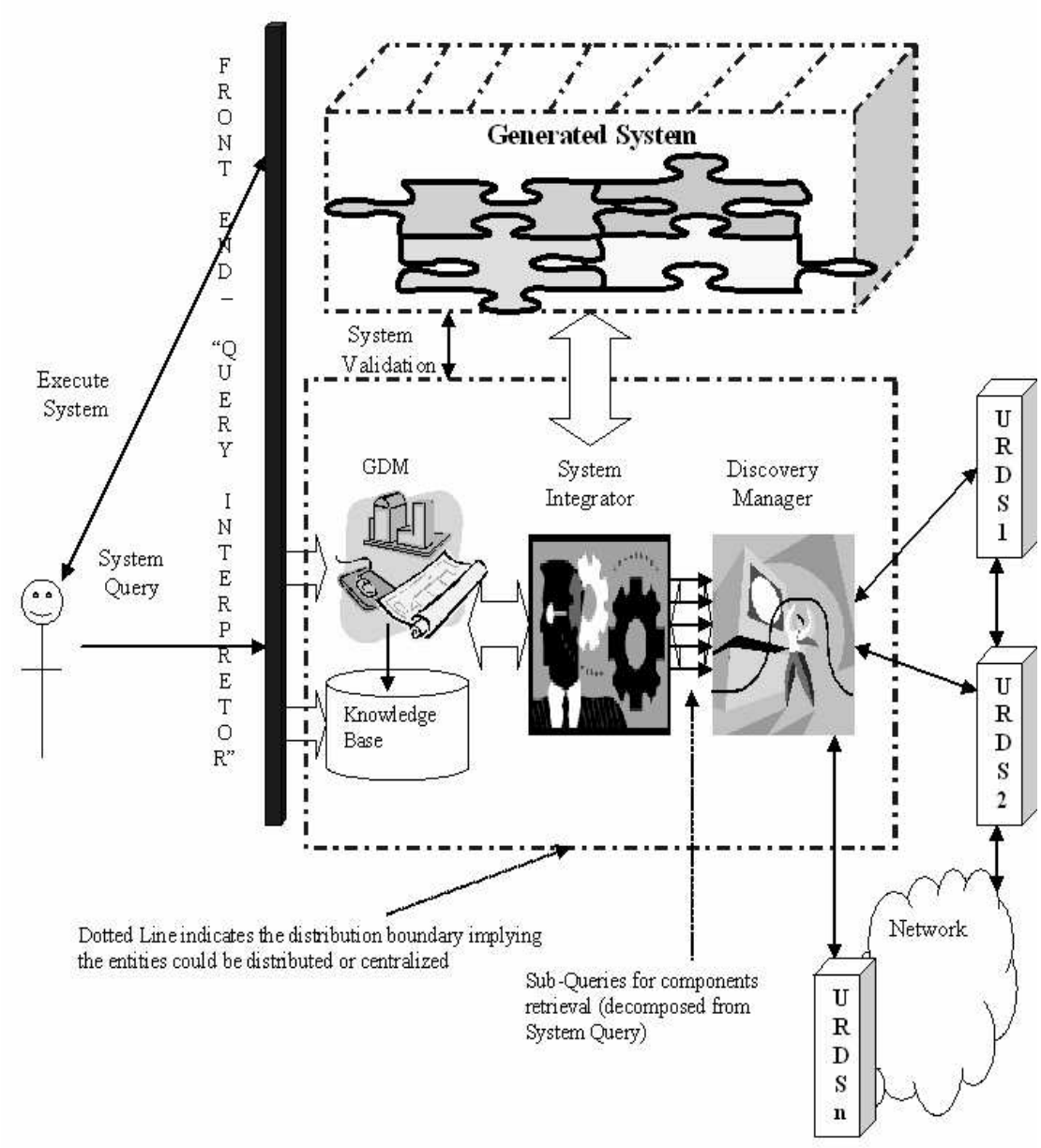


Figure 6.2 Participation of the Discovery Manager within the UniFrame

The figure shows the participation of the Discovery Manager within the UniFrame paradigm, with respect to the UniFrame discovery process. The entire process of handling the query was explained in brief in Chapter 2. In Figure 6.2, the main focus is the contribution of the Discovery Manager in handling the discovery process. The figure shows that the user submits a system query to the UniFrame system which then gets interpreted into a standard form, XML or some other machine readable format based on TLG, by a query interpreter. The query is then decomposed into queries for the individual components using the GDM (UniFrame QoS framework and the composition/decomposition model). These queries are then passed on to the Discovery Manager by the System's Integrator. The different discovery services that can participate in satisfying the queries are known to the Discovery Manager (through a protocol discussed later). Hence, the DM delegates each query to the known Link Managers. The Link Managers coordinate with each other to form a federated search space. After the results have been retrieved, the DM returns them back to the System Integrator. The System Integrator now continues with its task of building the distributed system using the Glue-Wrapper generator, out of the discovered components. The DM acts as the authority responsible for handling the entire discovery process across all the URDSs registered with it. Heterogeneous Link Managers also holds the problem of the communication between DM and these heterogeneous Link Managers. However as a part of this study, only the heterogeneity between Link Managers will be addressed while the interoperability between DM and the Link Managers is a future work in this direction. There can be multiple options associated with every different scenario of the architecture proposed in this study. The next section identifies all such situations for the above architecture and outlines the different options for each.

6.2 Different Points of Consideration

Following are the various issues that need to be considered while concretizing the architecture shown in Figure 6.2:

Entity of the URDS responsible to register the URDS with the DM: As indicated in Chapter 5, the Domain Security Manager authenticates all the principals of a URDS instance before they can operate within the URDS. It is the central authority for any URDS and starts up before any other entities within the URDS. Therefore, the architecture proposes that it should be the DSM which registers its URDS instance with the DM. There could be two scenarios to this approach:

- DSM registers the URDS on its start up marking the availability of associated URDS. Once the DSM is known to the DM, it can also be polled to query for the availability of the associated URDS. The credentials (discussed later) will be applicable at that point.
- A second solution is that instead of the DSM, it is the onus of the DM to build its repository of known URDS instances by polling the network to locate available instances of the URDS. In this case again the DM should establish a communication with the DSMs of the discovery services since it acts as a centralized authority for a URDS. The DM can then query it to establish it as authenticated URDS of the UniFrame system. The scenario is similar to the communication between the Headhunters and the Active Registries, but at a higher hierarchical level.

Credentials and Information of a URDS registered with the DM: DSM at the time of registration with the Discovery Manager should be able to claim that its principals (namely the HHs, ARs, QMs, AM etc) constitute a valid instance of a URDS. This requires some form of credentials to be provided by the DSM to the DM. The credentials serve two purposes: a) serve as identification for a URDS which aids in future communication between the DM and the URDS, and b) provides the DM with the various details of the URDS, namely the principals and associated QoS values. This fact is applicable in both the cases discussed above – DSM itself going to DM or the DM polling for the available services. Since every URDS is also a service with a well-defined role and its own set of resources, every URDS instance could hence have an associated UMM specifications. UMM specifications could include the details of various prinicipals

in the system, for example, the location of the access point (or the entry point, discussed later) of the URDS and the QoS values of the URDS. These set of specifications would be need to be however dynamic since the URDS has a dynamic participation of the entities and QoS values could build or vary over a period of time and performance. This area is not investigated in this thesis and can be a possible direction for future work. Possible credentials for DSM and its URDS could be:

```
Class Credentials
        {               string      UmmSpecURL;
                        string      DSMid; → checked to see if it is a valid DSM
                          ……<other parameters>
        }
```

UmmSpecURL – location of the UMM specification of the URDS instance of the DSM.

Entry point to a URDS instance: Since the QM and LM of the URDS handle the propagation of the query, there could be multiple access points to it. In case the QM is the entry point, it can collect the results for the query from the URDS it is associated with and then based on the policies put in place; it can propagate it to a LM for getting a different set of results. Or the query could also enter an instance through the Link Manager and then be handled by the LM based on the policies in place. However, multiple access points pose the overhead of ensuring the entry of legitimate users to use the discovery service.

One possible solution to the question could be the return parameter of registration by the DSM. The return parameter will be a unique token serving two purposes:

1. Authentication of the DSM as a representative of a legal instance of URDS based on the credentials submitted.
2. Establishment of the Discovery Manager as a valid client of the URDS. Hence, it can now pass the query either through the LM or the QM as long as it had made its token available to the DSM. DSM should then replicate the tokens with all its entry points.

Protocol for query propagation by the DM: The experimentation for this thesis has assumed that the DM will maintain a list of the instances of the URDSs which it finds or registers with it, whatever is the case. The maintenance of this list can be carried out in the following ways:

1. It could be maintained as a central repository list by the DM. However, in future if there are multiple DMs which can be contacted by the system integrator, then the list replication becomes a challenge in this case. Once the system becomes large there is a need to avoid inconsistency in the values the list contained at different DMs. Hence, a replication protocol needs to be employed in order for different DMs to have the knowledge of different URDSs that register with different DMs across the network.
2. Every DM could have a selected number of URDSs on their lists. This selection could be made on a premise such as maintaining the list only for the first "n" number of URDS that register with it. Other discovery services however would be rejected and could be directed to other DMs.

Once the list is maintained by the DM, the next issue is as to what protocol should be employed to propagate query to these URDSs on the list. Options for these are:

1. The DM can just pick up a random URDS on the list and then pass the list of the remaining services to this URDS for further propagation. However, again at a very large scale, there is always an overhead associated with passing at entire list for every query the DM receives.
2. With every query received by the DM, the DM picks up a URDS (again at random or the first one in the list) from the list it maintains and then passes only the query ahead to a URDS. All the URDS instances are periodically passed on a list of the URDS known to the DM. And on the event of receiving the query, the DM just needs to pass the query. This can lead to the overhead to network resources usage required for the periodic multicast of the list of LMs. However, this overcomes the overhead of passing the entire list of LMs with every query received by the DM. If a system is to be composed of a large number of

components, the overhead of passing the list of all the known LMs with each query could be large and hence can be avoided by this option.

3. The DM does not need to pass any kind of list of the existing URDS on the network. The onus could be on the LM of each URDS since it has to link to other URDS instances. Once the DM has passed on the query to one LM (choosing from the list it maintains), the LM there on takes the responsibility for the propagation of the query (addressed by the next question).

Protocol for a LM to know the existence of other LMs on the network: The question has many manifolds. The answer depends on as to how the query propagation is actually started by the DM in the first place. The different scenarios possible are:

1. Knowing it through the DM
   i. List of all the LMs knows to the DM is passed with every query propagated by the DM to the LM. This communication pattern is employed and experimented with when the QM contacts the HH with a query [MYS04].
   ii. Or the list is passed periodically from the DM to all the LMs. This is the scenario that will be implemented for the current scenario.
2. Self-acquaintance

   Every LM spools the network periodically to acquaint itself with the other existing LMs on the network. The LMs which respond to this broadcast get added to the known LM list of the querying LM. This communication pattern has been adopted for the current adaptation of communication between the headhunters and the active registries.

There could be multiple methods of learning about acquaintances. One of the algorithms is discussed in [PEN01]. It proposes a method for an entity to automatically learn the "best" acquaintances on the basis of the past experiences of interaction. This method is based on a reinforcement machine learning algorithm, called Pursuit Algorithm [THA85, MUK89]. A learning experiment for the entity takes place when its policy requires the propagation of the query to a remote entity. At that point, it chooses the

entity from the list of known remote entities based on a ranking which it assigns to them from the past experience; based on sampling set of vectors based on Pursuit Algorithm. For more information on Pursuit Algorithm, the reader is directed to [THA85, MUK89]. When the results of a query arrive, they are chosen on the basis on certain criteria:
First Arrival: the results which arrive the fastest are the "best", i.e. the entity which can return the results for a query the fastest or is geographically the closest will give the "best" results, and,
Classification quality: The thesis outlines a principle of choosing the best result in terms of the quality of the results – in terms of the best matching criteria. The criteria can be applied to the current context as the "Number of results returned for a given query".

Based on the above criteria, there could be two values associated with every query response - t_{th} and n_{th} - threshold values for the response time and number of matching components respectively. Once the "best" set of results has been chosen, the corresponding LM (or multiple LMs) is/are given a reward of +1 in the ranking of the LMs. Else if the LM fails to deliver results above the threshold values, the ranking is set by -1. Every time a query is propagated, the LMs are queried based on their ranking, or in other words the past history. [PEN01] shows that communication with remote agents based on a similar type of past experience yields a higher performance than other communication mechanisms where the communication was done on a random basis or any other such mechanisms.

Protocol for query propagation between the URDS instances:

1. Results are collected by the LM to which the query is passed in the first place by the DM. This LM, called the Parent LM, can query the other LMs known to it using one of the mechanisms mentioned above. The "Parent LM" and the rest of the LMs spanning the same search space (i.e. registered with the same DM), then form a propagation tree in which the query propagates from the root node ("Parent LM") to the leaf nodes. Results are finally returned back to the DM the root node LM. This

propagation technique has been experimented with the query propagation between the HHs and has been discussed in Chapter 5. There could be two options for this case

- Parent LM collects results from all the LMs known to it. However, this has an associated overhead in terms of performance. In case of a large number of LMs known to the Parent LM, the process might not be efficient to extend the search to such a large scope and can retrieve better results if the search is scoped by some parameters such as history of the known LMs, discussed next.
- Results could again be collected from different discovery services by one parent LM, as in the above case, but this time the LM collects the results only from a subset of the known LMs, based on the known history of the LMs. The algorithm for collecting results from the remote entities based on the past experience and history has been discussed earlier and more details can be found in [PEN01].

2. Every LM who receives the query searches its own URDS for the results and then passes the query to the one of the other LMs in its list of known LMs. The next LM in the chain then takes over the propagation of the query. The process repeats itself till all the LMs in the list are exhausted. Since every LM associated with a DM knows the location of the DM, the last LM in the chain of query propagation can submit the results for that query to the DM. The mechanism requires a book-keeping methodology in order to ensure that there are no loops in the propagation of the query. For example, a possible solution can be that the DM passes the list of the known LMs to the first LM along with the query. After a LM processes the query in its own URDS, the LM location is removed from the list and the query with the list of the remaining LMs is propagated to the next LM. The process continues till the list is exhausted. However, the approach incorporates the overhead of passing the list along with the query every time on a query propagation.

6.3 Chosen Scenario for Experimentation

From all the above mentioned cases, for the current prototype implementation, the following sequence of events have been selected from a perspective of experimentation

- Discovery Manager acts as a centralized location of all the registered discovery services. Hence, in the case of multiple DMs (targeted by a single System Integrator or by multiple System Integrators), the system can be further scaled and/or scoped. This could imply that the components retrived from one DM and one SI, lead to the formation of one sub-system which can then be composed to form a larger system made up of multiple smaller distributed systems. Each DM would be responsible for the discovery of components needed to build one single sub-system.
- UMM specifications for a URDS: For the current implementation, only the ID of the URDS would be filled in. In addition, the domain name (as Discovery) and registration entity as location of the Discovery Manager is also included. The ID would be the location of the access point (to be discussed in next question) of the URDS and can be used by the Discovery Manager to route the queries to that URDS. The protocol of communication or propagation of query will be discussed later.
- Link Manager acts as the sole entry point to a URDS. As a future work; even though now the QM can be queried to retrieve the results of a query through a client, the necessary security mechanism needs to be ensured that such a client can be authenticated before the query is processed. Right now the authentication is needed only at the end of the LM before the query processing can be initiated. The LM receives a token from the DM along with the query, and it can only become the recipient of the token if its location was made known to the DM by the DSM upon the URDS registration.
- DSM starts up and authenticates its URDS with the DM. It passes the credentials (URDS UMM specifications URL and its DSM id) to the DM. The URDS UMM specifications have the <LinkManager> as one of the tags. Hence, when the DSM registers, the DM parses the <LinkManager> to get the location of the LMs and stores it in its database of the known LMs.
- DM periodically passes the list of all the registered LMs to every LM. This reduces the overhead of passing the entire list of the LMs with every query

received by the DM. For example, a single system query could be decomposed into a large number of components' queries. However, as the DM has already been periodically propagating the list of LMs to every LM, the queries need not be propagated along with the list.

- LM receives the list and refreshes its database of known LMs with the new list which it periodically receives from the DM. Also, periodically the DM will be spooling the known LMs to see if they are alive. This could also have been done by the LM but since the DM holds the responsibility of propagating the list of the known LMs, in this scenario the DM also takes in the responsibility to keep the list updated.
- Upon receiving a query from the SI, the DM picks up a random LM (prototype: the first one) from the known LMs list and passes the query to it.
- Protocol for propagation of query within the LMs: One LM checks the policy and then propagates it further to other LM. Right now since there is only one DM, all the LMs are aware of the location of the DM supplied to them by the DSM when the LM authenticates itself with the DSM on startup. Also, there could also be a policy associated with the query when it comes from the DM. The DM can attach this policy with the query based on the time sensitiveness of the result retrieval of the query. For example, if the DM is informed by the SI that the result retrieval time for a particular query is time sensitive, then the DM can associate a max limit of components to be retrieved for a query. When a LM receives a query, it first checks the number of components retrieved for that query (since the previous LM passes it to the LM), and if needed it executes the query or else passes the query to the next LM in the list. When the LM tries to contact the other LM, it – "Linking Dock" – placeholder of the connector (part of it since the other part will be in the Lining dock of the other LM) instantiated by the GWG when the other LM is from another component model.
- The last LM in the list of processing queries submits all the results to the DM.

6.4 DM Architecture

Every system query $q_{s,}$ in a standard form is decomposed into a set of individual queries,$\{q_{c1}, q_{c2}, q_{c3}…..q_{cn}\}$, for individual components required for the system. The DM is handed all these q_c by the system integrator. The DM implements a certain functionality to handle the discovery of the components based on q_c. These functions have been outlined in Section 6.4.1 followed by the policies that the DM implements to manage the discovery process. Section 6.4.3 discusses the algorithms that constitute the functionality of the DM.

6.4.1 Functions

The DM supports the following set of functions. The algorithm for each of the functions will be discussed in Section 6.4.3.

Register URDS Instance: The function of DM allows the DSM of an URDS instance to register it service with the DM and participate in query handling.

Refresh List of Known LMs: The DM refreshes its list of known LMs by periodically contacting the known LMs and checking if they are alive. The LMs which do not respond are cached as dead LMs. If the URDS DSM attempts to try and re-register within a particular interval of time, the LM is then restored to the list of active LMs. This requires the fault tolerance on the part of the DSM to be aware of its LM existence and then try and re-register the URDS with the DM.

Periodically propagate List of known LMs: The DM periodically propagates the list of its known LMs to all the LMs. This function serves as a means to make the LM aware of the other LMs to which it can communicate with in order to achieve query propagation.

Initiate the Discovery of query q_c: For every q_c, the DM initiates the discovery process between the URDS instances registered with the DM.

Submit Results for query q_c: Since the LMs employ a *chain protocol* of handling the propagation of every q_c, the last LM in the chain submits the results to the DM. the DM provides the submission of the results which are identified with the help of the query id accompanying the results.

6.4.2 Policies

The DM needs to employ certain policies which administrate the propagation of queries between different discovery services. For now, only one policy to limit the extent of search has been outlined.

SearchExtentLimit Policy: $N_{max}|q_c$ can be specified with every component query q_c to limit the number of services to be returned for every qc. This number can be decided on the time sensitiveness of each query q_c. The policy acts as an optimization policy since it avoids a long delay in retrieving the information for every q_c and hence the system generation process can be carried out in a timely manner.

6.4.3 Algorithms Supported by DM

6.4.3.1 Register URDS Instance

Figure 6.3 depicts this algorithm. On the start up of a URDS instance, the DSM contacts the DM to register the instance of the URDS. The credentials consisting of the UMM specifications of the URDS are submitted to the DM. The dynamic nature of the UMM specifications requires some form of protocol to refresh the values of the UMM specifications at end of the DM. However, the scenario has not been explored by the

study as it lies out of the scope of the thesis. Also, there needs to be a way to claim from the UMM specifications prove a valid instance of the URDS. This could be checked on the basis of the QoS of the UMM specifications registered or some other form of mechanism to ensure validity. The topic however lies out of the scope of the study.

```
REGISTER_URDS
IN: URDS_Credentials
OUT: Token
IF (URDS_ID does not exists in Database)
      Add URDS specifications
      Return Valid Token
ELSE
      Registration Failed
      Return Empty Token
END IF
END REGISTER_URDS
```

Figure 6.3 Algorithm: Register URDS instance

The LM location is then parsed by the DM from these UMM specifications and added to its list of its known LMs.

6.4.3.2 Refresh the List of Known LMs

The DM periodically contacts all the LMs in its list to verify if they are alive. The list is updated with the statsThe algorithm for the bookkeeping is depicted in Figure 6.4. The DM contacts every LM periodically and if the LM responds to its request it is marked as alive and is available for future query propagations. If the test however fails, the LM is removed from the list of alive LMs and then shifted to the list of past LMs. In the next phase, each of the past LMs is now again checked to check the timestamp on the LM. If the DSM of the corresponding URDS has attempted to re-register the URDS and the LM with the DM, the LM would be removed from the list of Past LMs to the list of alive LMs. However, if the timestamp is greater than the Tmax timelimit, the LM is declared to be dead and the DSM of the URDS now needs to register.

```
REFRESH_LIST_KNOWN_LMs
    For each LM in ListOfKnownLMs
        If ContactLM.Exists = True
            Update LM with latest contact information
        Else if ContactLM.Exists = false
            For n=0 to n_retry,  Retry contacting LM
                                 //Retry for a retry limit, n_retry
            If number of attempts = n_retry,

            Remove LM from the list of alive LMs
            Cache LM to the list of Past LMs //the LM needs to re-
            register with the token it was given by the DM during
            the first registration
    End For
    For each LM in PastLMs
        If timestamp (last modified time) > Tmax
            Remove the LM from the database        //if a LM
                was once declared as dead, and it did
                not respond to re-register within a time
                interval of Tmax, then remove it from
                the list of cached LMs. Caching helps in
                giving the same token with a renewed
                time stamp to the LM which contacts
                again
    End For
END REFRESH_LIST_KNOWN_LMs
```

Figure 6.4 Algorithm: Refresh list of known LMs

6.4.3.3 Propagate the List of Known LMs to Every LM

The algorithm is depicted in Figure 6.5. The DM uses the algorithm to propagate its list of known LMs to all the registered LMs on a periodic basis. For each LM in its list, it contacts the LM and if it responds to its request, it updates the information of the LM. If the communication fails to be established, it attempts a retry for a certain value n_{retry} and even if the communicatin fails, the LM is marked as dead by removing it from its list of known LMs and caching it in the list of past LMs. The LM can be put back in its list of alive LMs by the use of the REFRESH_LIST_KNOWN_LMs algorithm as shown in Figure 6.4.

```
PROPAGATE_LIST_KNOWN_LMs
        FOR each LM in ListOfKnownLMs
            If ContactLMExists = True
              LM.UpdateKnownLMs
                                  //invoke the "UpdateKnownLMs" function on
                                  the contacted LM.

            Else If ContactLM.Exists = False        //unable to contact
              For n=0 to n_retry,  Retry contacting LM  //Retry for a retry limit, n_retry
                  If number of attempts = n_retry,
                          //Declare the LM as dead
                          Remove LM from the list of alive LMs
                          Cache LM to the list of Past LMs
                                                  //the LM needs to re-register
                                                  with the token it was given by
                                                  the DM during the first
                                                  registration
        End FOR
END PROPAGATE_LIST_KNOWN_LMs
```

Figure 6.5 Algorithm: Propagate list of known LMs

6.4.3.4 Initiate the Discovery Process for q_c

This algorithm is employed by the DM to initiate the process of query propagation on the event of receiving a query from the system integrator. It receives the component query, qc, along with the ID of the query given by the system integrator. It adds the query to its list of current processing queries and sets the status as "New". The query is then propagated to a random LM from its list of known LMs. Once the query handling is successful, the status is set as pending and updated when the results are submitted by the LM. Figure 6.6 depicts this algorithm.

```
MANAGE_QUERY_PROCESSING
        IN: q_c, queryID
                                  //component query qc has been assigned a unique
                                  query ID by the system's integrator
            Add the queryID to the list of CurrentSystemQueries
```

Figure 6.6 Algorithm: Initiate discovery process by the DM

```
                Update Status = New
                Pick a random LM from ListOfKnownLMs
                Call LM. HANDLE_QUERY_PROPAGATION
                Update Status=Pending for QueryID
END MANAGE_QUERY_PROCESSING
```

Figrue 6.6 Continued

6.4.3.5 Deposit Results – Invoked by the LM Submitting Results for a Query

This algorithm is followed by the DM when a LM submits its results for a given query. The DM checks the QueryID of the query. If it exists in its database of CurrentSystemQueries, the results of the query are stored and the status is updated from "New" to "Processed". The results are then returned back to the System Integrator. However, if the query Id does not exists with the DM, the DM discards the results since the query was not routed through this DM. The algorithm is depicted in Figure 6.7.

```
        SUBMIT_QUERY_RESULTS
        IN: QueryID, Results
        IF (QueryID exists in CurrentSystemQueries)
                        //"CurrentSystemQueries" is a store of component
                        queries which are currently in process by the URDS
                        system under the control of this DM
                Update the status of the query as "Processed"
                Update the query results "Results"
                Return <Results, QueryID>  to System Integrator
        ELSE
                Discard the results
                        //the results do no belong to this DM
                        //may be the query belongs to another client or DM
END SUBMIT_QUERY_RESULTS
```

Figure 6.7 Algorithm: Submission of results to DM

6.5 LM Architecture

The LM implements a set of functions based on the chosen scenario of query propagation. The following sections outline the implemented set of functions by the LM followed by the policies and the algorithms incorporated as part of the functions.

6.5.1 Functions

InstantiateLM: Prepares the different threads of the LM.

PerformSearch: Depending on the policy for the LM, the LM hands over the query received to the QM belonging to its own URDS. This function provides the necessary functionality.

UpdateKnownLMs: On receiving the list of LMs from the DM, the LM refreshes its database of known LMs to be used for propagation.

PropagateQuery: The function defines the protocol to propagate the received query from the DM or the previous LM to the other known LMs.

6.5.2 Policies

The policies govern the behavior of the LM at runtime. Policies are provided as name-value pair and can be grouped into two categories: 1) Policies that govern the extent of the search scope and 2) Policies that determine the functionality applied to a query execution. The examples of search scoping policies are i) an upper bound on the number of services to be searched from, ii) an upper bound and lower bound on the number of services to be returned.

The aim of the chapter is to study the linking of the discovery services; however, since the underlying motive continues to be to explore the issue with respect to the .NET component model, there exists a need to address the issue of with respect to the .NET component model. Hence it becomes important to study the linking of discovery services with the issue of interoperability between two heterogeneous component models. The following section explores this issue and outlines an approach adopted for the case-study. Since Chapter 5 provided an analysis of the .NET and Java RMI URDS, it is best suited

to experiment the interoperation with respect to URDSs belonging to the two component models. Section 6.6 provides the necessary details.

6.5.3 Handling Interoperability with a Heterogeneous LM

The proposed LM handles interoperability with a LM in a component model different from its own, by means of a connector which encapsulates the interoperability mechanism and is generated with the help of the GWG. The GWG uses the connector model which has been discussed in Chapter 3 and illustrated in Section 5.3.2, Figure 5.8. Since every LM during the process of query propagation, may need to connect to a LM in the other component model; there exists a need for the LM to provide for a "Linking Dock" where the connector can be instantiated by the GWG. The Linking Dock specifies the environment where the connector will be instantiated by the GWG and its specifications are provided to the GWG partially by the GDM and partially by the LM. The process is now discussed and is based on the work by [BUL00].

6.5.3.1 Connector Generation for Link Manager

The connector model was discussed in Chapter 3 and then also briefly in Chapter 5. This section briefly discusses the generation of a connector for the LM which enables connecting heterogeneous ICBs. The work in [BUL00] proposes a model for the automatic generation of the connectors and forms the basis of the approach discussed in this section.

The typical lifecycle of a connector has been depicted in Figure 6.8.

Design: Since the connector frame specifies a black-box view of a connector, it represents the most generic form of a connector specification and constitutes the first stage in the connector lifecycle. The Connector frame only specifies the connector endpoints (roles) which may be generic or may be bound to a particular interface. In the case of LM, since the interface is known by the GDM as part of the Abstract Component

Model, the connector frame for the LM can be bound to its particular interface and its specifications for the LMs can be stored in the knowledgebase.

Development: Since the connector frame is only a black-box view of the connector, it can be implemented by multiple architectures. Hence, once the connector frame has been defined, the next step is the definition of the architecture of the connector. The architecture for the LM could be predefined since the LM is predefined as part of the UniFrame system. The architecture specifies the type and instances of the primitive elements which form the connector implementation and the bindings between them. Since the architecture is inherently distributed, every primitive element is defined in a way to be contained within a distribution unit. The explicit specification of distribution boundary is also included in architecture specifications. The architecture is specified during the development stage and the specifications are flexible to be extended during the deployment. Since the architecture of the LM can be predefined, the architecture for the connector of LM can be a part of the knowledgebase.

Deployment: It is during the deployment stage that appropriate implementations are assigned to the primitive elements of the connector. Hence, the connector's interface can be further modified (if need be) and behavior can be modified by changing the implementations of the primitive elements. The implementations are generated on the basis of the deployment descriptor given to the GWG by the LM. For example, the deployment descriptor can consist of the <Client_LM_Component Model, Server_LM_ComponentModel, Client LM's Linking Dock Specifications>where

Client_LM_Component_Model = Component Model of the LM which requests the instantiation of the connector

Server_LM_Component_Model = Component Model of the LM to which the client LM wants to connect to.

Client LM's Linking Dock Specifications = specifications of the *Linking Dock* of the client LM in which a connector unit should be instantiated by the GWG.

Hence, on the basis of the deployment descriptor and the pre-defined connector architecture for the LM connectors, the GWG generates the necessary connector and returns to the client LM a reference or handle to it.

The architecture of the connector generator based on the connector lifecycle depicted in Figure 6.8 has been outlined in [BUL00]. In line with the idea of two independent abstractions, namely the primitive elements and the connector architecture, there are two generators which handle the task of connector generation. These are connector generator (CG) and element adaptor (EA). The generator is made up of several modules which can be generic or implement generation of one or more connectors. They can implement it using predefined code for each connector, or they can use connector architecture for more automated generation. The generator modules use the element adaptor for generating the building blocks of the primitive elements. The generator modules can also house various vendor-specific connector mechanisms which can help generate primitive elements incorporating these mechanisms. This describes the way in which the bridge used for the experimentation for the case study can become a part of the glue-wrapper generator. [BUL00] provides further details on the proposed approach of the connector-generator. The project also implements a case study incorporating the interoperability between Java RMI and CORBA. However, the prototype validates interoperability across different component models but same language, Java. As part of a series of personal email communications [EMA03b], Tomas Bures, lead person of the project in [BUL00] confirms that interoperability between .NET and Java RMI model has not been experimented in the project and the proposed model of automation of connectors. Section 6.6 attempts the interoperability between these two component models in the context of linking discovery services.

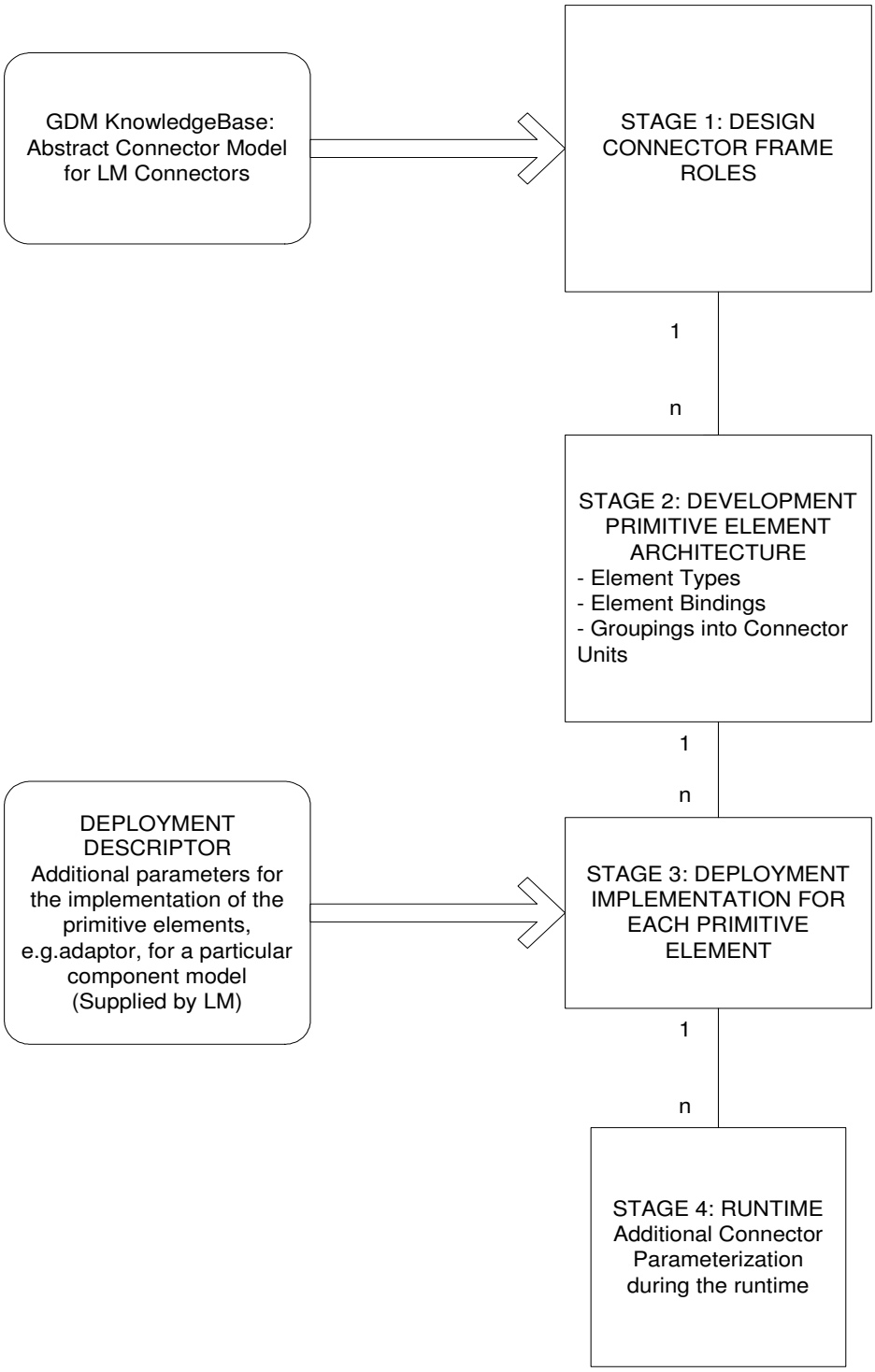


Figure 6.8 Connector lifecycle

6.5.4 Algorithms Supported

This section lists the various algorithms that are supported by the LM with the chosen architecture.

6.5.4.1 Algorithm for LM Initialization

The algorithm initializes the entire LM process for its start-up. This can include some input from the ICB configuration manager – such as setting the policies etc. Figure 6.9 depicts this algorithm.

```
LM_INITIALIZATION
        ACTIVATE   LM_DM_LISTENER_UPDATE_KNOWN_LMs
        ACTIVATE   LM_LINKER_DOCK
        ACTIVATE   LM_QUERY_PROPAGATE_SERVER
        ACTIVATE   LM_CLIENT_REQUEST_HANDLER
END_LM_INITIALIZATION
```

Figure 6.9 Algorithm: LM initialization

The algorithm to initialize the LM begins with initializing the DM listener thread so that its database can be updated with the registered LMs of the DM. This needs to be the first step since at any time a client request comes in, the database of the LM's known LMs should be updated and the LM should be in a state of propagating the query to other LM depending on the policy. Secondly, the LM initializes its "Linking Dock", to prepare for instantiation of the connector by the GWG in case the LM to be contacted is heterogeneous. And now the LM initializes the query propagation service. After all these services have been initialized, the LM now initializes the request handler to accept any incoming requests – by the DM.

6.5.4.2 Algorithm for Updating Database of Known LMs

The set of operations are performed periodically by the LM upon receiving the list of LMs from the DM. It is the responsibility of the DM to keep the list updated. Hence, upon the receiving the list from the DM, the LM checks its list for every entry of the LM and if it exists, it is updated with its latest information from the DM else it is inserted. Figure 6.10 outlines the algorithm.

```
LM_DM_LISTENER_UPDATE_KNOWN_LMs
        IN: ListOfKnownLMs_DM
        OUT: Success/Failure
                For each LM_Location in ListOfKnownLMs_DM
                        If LM_Location exists in MyDatabase
                                Update LM_Location
                        Else
                                Insert LM_Location
                End For
END LM_DM_LISTENER_UPDATE_KNOWN_LMs
```

Figure 6.10 Algorithm: Algorithm for updating list of known LMs

6.5.4.3 Initialize the Linking Dock

The linking dock is represented in terms of an object of the type "Linking Dock" which supports the standard interface of the Link Manger enabling other LMs to communicate with it in a standard form. The class in instantiated with the deployment descriptor of its linking dock and then registered with the GWG which can later instantiate connectors using this reference (Figure 6.11).

```
LM_LINKER_DOCK
        //Initialize the environment for instantiating connector
        Linking Dock LD  = new Linking Dock(Deployment Descriptor)
                //the class acts as the representative of the Linking Dock of the LM
```

Figure 6.11 Algorithm: Initialization of the Linking Dock

```
                                        //it has a predefined interface of a
                                        standard LM and is also used by the
                                        GWG for future communication for
                                        connector instantiation
    GWG.registerLinkingDock( LD)
                                        //register the linking dock with the
                                        GWG providing the necessary details
                                        of its linking dock to the GWG in
                                        form of a  deployment descriptor of
                                        the linking dock (Figure 6.8). This
                                        allows the GWG to instantiate the
                                        necessary connector in this LD in the
                                        future.

END LM_LINKER_DOCK
```

Figure 6.11 Continued

6.5.4.4 Algorithm for Performing Search in Own URDS

The algorithm to collect the results from the LM's own URDS. The LM passes the query to the QM and from there on the discovery process is carried out by the URDS, as discussed in some of the earlier sections (Figure 6.12).

```
PERFORM_SEARCH
IN: query
OUT: Results
    Results = myQM.PerformSearch(Query, QueryID)
                    //Get Results by passing query to own QM
END PERFORM_SEARCH
```

Figure 6.12 Algorithm: Handing the query for search in own URDS

6.5.4.5 Algorithm for Handling Query Propagation from the Client

The inputs to a LM at the point of entry of a query are, the query, the ID of query which can also be retrieved from the query object as well, Nmax: is the maximum number of components specified by the DM for a particular query and Rprev is the

number of results obtained from the previous LM in the chain. The LM is configured with two policies: a) Policy_Number_Components and b) Polic_Extent_Search_Scope. a) is a local policy configuration of the LM which defines the minimum number of components that should be retrieved from the search results of its own URDS. b) can either have values as “Always” or “Null”. “Always” implies that irrespective of the value of Rprev, the LM propagates the query to its own URDS and the results are added to the list of results from the previous LM. Since, the algorithm incorporated a lot of details it has been depicted in the flowchart of Figure 6.14 and outlined by the following pseudocode (Figure 6.13).

```
LM_CLIENT_REQUEST_HANDLER:
HANDLE_QUERY_PROPAGATION
IN: Query, QueryID, N_max, R_prev,
KNOWN:Policy_Number_Results , Policy_Extent_SearchScope
OUT: resultTable
   IF QueryID Exists in ProcessedQueries        //ProcessedQueries hold the
                                                 IDs of the queries processed by this
LM
     PASS_QUERY_OTHER_LM
  END IF
  ELSE
    CHECK Policy_Extent_SearchScope      //policy can be specified as an XML
                                         file for the LM  and hence even if the
                                         policy changes, it should be effective
                                         with every new query propagation the
                                         LM undertakes, after the changes have
                                         been made
        IF Policy_Extent_SearchScope = “Always”
                                          //always get  results from own  URDS
            Results=EXECUTE LM_PERFORM_SEARCH          //get the results
                                                       from own
                                                       URDS
                                                       irrespective of
                                                       the current
                                                       value of R_prev
```

Figure 6.13 Algorithm: Propagate query to other LMs

```
                IF Results >= Policy_Number_Results          //own policy
                        IF Results >= Nmax                   //Nmax given by DM
                        DM.SUBMIT_QUERY_RESULTS
                        ELSE
                        PASS_QUERY_OTHER_LM
                 END IF
                 ELSE
                        PASS_QUERY_OTHER_LM
                  END
          END IF
          ELSE IF Policy_Extent_Search_Scope = "Null"
                  IF Rprev >= Nmax
                       DM.SubmitResults
                 ELSE
                        Results=EXECUTE LM_PERFORM_SEARCH
                                                              //Search own
                                                              URDS only if
                                                              Nmax not
                                                              satisfied since
                                                              Policy_Extent_
                                                              Search_Scope is
                                                              Null
                        IF Results >= Policy_Number_Results   //own policy
                        IF Results >= Nmax                   //Nmax given by DM
                              DM.SUBMIT_QUERY_RESULTS
                        ELSE
                               PASS_QUERY_OTHER_LM
                        END IF
                        ELSE
                             PASS_QUERY_OTHER_LM
                        END
                 END //for policy check null
          END ELSE IF
          END
          END
          END ELSE IF
END HANDLE_QUERY_PROPAGATION
```

Figure 6.13 Continued

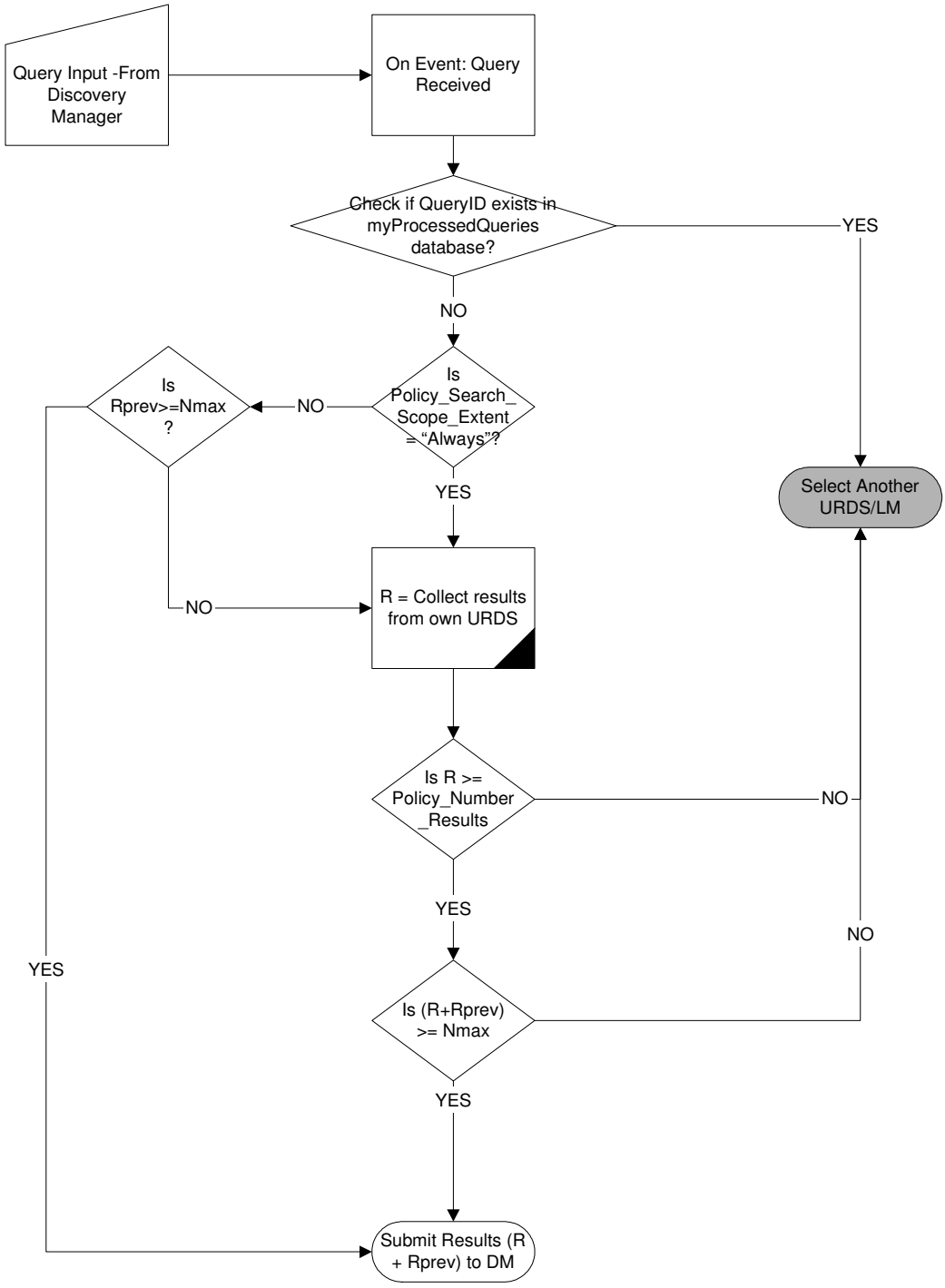


Figure 6.14 Query handling by the LM

6.5.4.6 Pass the Query to Other LM

The algorithm is employed by the LM at the point of query propagation to the other LM. This algorithm employs the contacting the GWG in case of detecting the heterogeneity of the other LM to be contacted. The algorithm shows that when a LM detects that the other LM is heterogeneous (the component model information of the LMs is passed by the DM to all the LMs along with the list of their locations. The DM derives this information from the UMM specifications of the URDSs registered with it), it contacts the GWG with parameters as, its own component model and the component model of the other LM. Since the LM has registered the deployment descriptor of its Linking Dock with the GWG, the GWG uses the combined information to instantiate the connector returning back a reference to the LM to invoke. In case of homogeneous LM, this can be bypassed. Figure 6.15 outlines the algorithm.

```
LM_QUERY_PROPAGATE_SERVER
    Check the ListOfKnownLMs
    LMnext = Pick a random LM (or based on some ranking)
    IF LMnext.model NOT Equal LM.model
        ConnectorReference=ContactGWG
        (LM.Model, LMnext.Model)
        ConnectorReference.HANDLE_PROPAGATE_QUERY
                                        //the connector is
                                        parameterized to the
                                        same interface as the
                                        LM
    END IF
    ELSE
        LMnext.HANDLE_PROPAGATE_QUERY
    END
END LM_QUERY_PROPAGATE_SERVER
```

Figure 6.15 Algorithm: Pass the query to other LM

6.6 Experimentation

The experimental prototype for the study consists of spanning the search space of the URDS across more than one instance of the URDS. As was mentioned in Chapter 5,

there are now two adaptations of the URDS which could be a part of the experiment. These are Java RMI and .NET Remoting. An architecture consisting of the DM, .NET URDS (incorporating the .NET LM) and the Java RMI URDS (incorporating the Java RMI LM) were experimented with to propagate a component query from one instance to the other instance. Results were collected from both the discovery services and returned back to the DM. Since the two instances are heterogeneous in nature, a connector incorporating a .NET-Java bridge was used to achieve the interoperability. The connector has a simple architecture with the structure as shown in Figure 5.8 and discussed in Section 5.3.2. That is, at the time, the connector architecture only consisted of the primitive elements, stub, skeleton and the adaptor. No other primitive elements have been included to support any QoS-related or other features such as logging, interception etc. Though such features can be incorporated in the future as part of the knowledgebase which will be used by the GWG define a different architecture of the connector for mediating between the LMs. The Interface of the connector supports the LM interface - I_{LM}. The I_{LM} right now supports only the method for a LM to pass the query to the other LM. Hence I_{LM} consist of the method "handleQueryPropagation(Query, <ResultsCollectedTillNow>, Rprev=count of all the previous collected results, Nmax). In the case of heterogeneous component models, the communication between LMs is carried out through a connector with a bridge, whereas in the case of homogeneous discovery services the communication is devoid of the connector. In order to study the effect of this difference in the performance of the federation of the URDS, two experiments were performed. These experiments can be divided as:

1. Homogeneous federation behavior: This experiment consisted of propagation of a query from a Java RMI URDS to another Java RMI URDS and collection of results based on the alternatives chosen in Section 6.3.
2. Heterogeneous federation behavior: The experiment consisted of federation across Java RMI and .NET URDSs with the same query propagation protocol.

These categories help to clearly distinguish between the natures of the discovery services, in terms of their component models and study the role played by heterogeneity in the federation. However, due to the technological difficulties faced in the

establishment of two one-way bridges on the same machine (for communication between the Java RMI LM and the .NET LM and .NET LM and Java RMI DM), the propagation of results was routed back though the same LM which is first in the chain. All the other details of the process however remained the same. The chain of actions is depicted in Figure 6.16 and 6.17 for the homogeneous and heterogeneous experiments respectively. The variables in the figure will be discussed in detail during the analysis of the experiment in Section 6.6.3. The next section now outlines the experimental set-up for the two experiments.

6.6.1 Experimental Set-up

The experimental setup consisted of the following individual set ups:

System Integrator and the DM: The System Integrator acts as the client to the system currently and initiates the discovery process by passing a set of q_c to the DM for each case of the experimentation – homogeneous and heterogeneous discovery services. The current realization of the UniFrame's system integrator [HUA01] uses the Java RMI component model. Hence, to keep this prototype extensible for future integration purposes, the system integrator has also been developed using the Java RMI model. The same holds true for the DM as well. Both the entries were deployed on Windows XP operating system and developed using the JavaTM 2 Platform, Standard Edition (J2SE) version 1.4 software environment. The hardware consisted of a laptop, Dell Inspiron 8100, P III processor.

.NET URDS: It consisted of the following .NET URDS entities: DSM, HH, QM, AR, .NET components and the .NET LM. All the entities are developed using the .NET Remoting model. The DSM, HH, QM, AR comprise the same URDS which was used for the experimentation of Chapter 5. These entities were deployed on Windows 2000 desktop, Dell Optiplex GX150, P III processor. The LM was deployed in the same environment as the system integrator and DM, mentioned above.

Java-RMI URDS: Consisted of the following Java-RMI URDS entities: DSM, HH, QM, AR, Java-RMI components and the Java-RMI LM. The DSM, HH, QM, AR and the Java-RMI components were developed using the Java-RMI model under the JavaTM 2 Platform, Standard Edition (J2SE) version 1.4. The entities were deployed on a Solaris machine hosting the UNIX operating system. However the file system used was on a UNIX-based server where all the files resided. Hence, the execution of the URDS system depicted higher retrieval times as will be indicated in the analysis. The Java LM was deployed in the same environment as the system integrator, DM and the .NET LM.

.NET-LM and Java-RMI LM Connector: For the purpose of the experiment the connector for the mediation between the two LMs has been manually crafted and placed in the linking docks of the two LMs. In the current prototype, since only one-way bridge was used (as explained earlier) for the propagation of the query from the Java LM to the .NET LM, the connector's distribution boundary was designed to the place the stub and the adapter in the linking dock of the Java LM and the skeleton on the .NET LM. The adapter comprised of the Ja.NET Runtime environment (as discussed in Chapter 2, Section 2.4.3, and Figure 2.7) and its interface was tuned to that of the LM with manual intervention. The LM communicates to the heterogeneous LM through the interface of its Linking Dock. Hence, the connector <stub, adapter, skeleton> was on the laptop hosting the LMs and the DM.

6.6.2 Experimental Use-Case

Based on the above mentioned outline, the two experiments were performed as depicted in Figure 6.16 and Figure 6.17. The system integrator submits a set of eleven queries to the DM in a sequence after an interval of few milliseconds. For every query received, the DM calculates the time it takes for it to receive back the combined results from both the URDS instances registered with it.

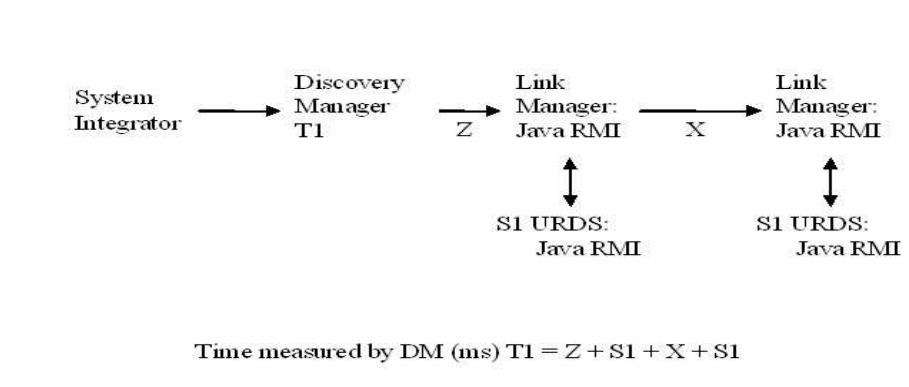


Figure 6.16 Homogeneous federation experiment

In experiment 1, Figure 6.16, the DM submits the query to a Java-RMI LM which then implements the "Always" policy for its Poilcy_Search_Extent and hence, propagates the query to its own Java-RMI URDS. Let S1 be the time taken by the Java-RMI URDS to retrieve the results for one query (measured at the QM of the URDS). Then the Java-RMI URDS propagates the same query to the next LM in its list which is homogeneous in this experiment. Based, on the same policy the LM retrieves results from its own URDS. Now since the list of known LMs is exhausted, the Java-RMI LM returns the results back to the previous LM and the results are submitted back to the DM. Since, in this case the second URDS in the chain is a replica of the first URDS, the time of computation for the same query can be denoted by S1 as well. Let the time it takes for the communication between the two URDS be denoted as X and between the DM and the first Java LM be Z.

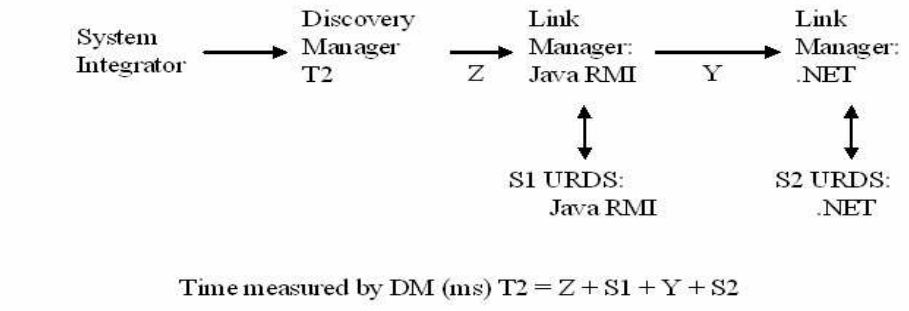


Figure 6.17 Heterogeneous federation experiment

Experiment 2, Figure 6.17, consists of the same sequence of the steps as experiment 1 except that the first Java-RMI LM now propagates the query to a .NET LM. In this case, since the two URDS instances are heterogeneous in nature, let the time it takes for the results to be retrieved from the first Java-RMI URDS be S1 (same instance as the one in the homogenous experiment) and by the .NET URDS be S2. Also, since the DM still passes the query to the same Java LM in the first place, let Z denote the time it takes for the DM-Java RMI LM communication. However, the communication between the two heterogeneous LMs is Java-RMI LM --- Connector --- .NET LM. Therefore, let Y denote the time taken for the communication in this case (which may or may be equal to X).

In either of the two cases, time taken for the results retrieval is calculated by the DM. And approximately, ignoring the small variables in the delays due to the network, the times will be equal to the sum of the variables identified in the above two use-cases. Hence, these values can be given by the following equations:

Homogeneous federation: T1 = Z + S1 + X + S1 ---------- (1)

Heterogeneous federation: T2 = Z + S1 + Y + S2 ----------(2)

The results for the two experiments measuring the values of T1 and T2 have been shown in the next section along with their analysis.

6.6.3 Results and Analysis

The results for eleven queries presented by the system integrator have been shown in Table 6.1. The values for the heterogeneous federation lie in the range of 17305 ms to 19097 ms. And the average value is taken as the value of T1=18116.27 ms. The table also shows the values retrieved for the homogeneous federation which lie in the range of 34420 ms to 36262 ms and average value amounts to T2=35241.73 ms. The difference in the values can be obtained by subtracting equation (2) from (1),

- ⇨ T1 – T2 = (Z + S1 + X + S1) – (Z + S1 + Y + S2)
- ⇨ 35241.73 - 18116.27 = Z + 2S1 + X – Z – S1 – Y – S2
- ⇨ 17125.46 = S1 – S2 + (X – Y) ---------- (3)

Due to the principles explained for the Ja.NET bridge in Chapter 2, Section 2.4.3, it was inferred that the bridge leverages the Remoting paradigm for the communication between the heterogeneous entities. Hence, in both the homogeneous and the heterogeneous federations, the communication is using a binary protocol, Java serialization for homogeneous and Remoting serialization for heterogeneous federation. In addition, since the heterogeneous federation employs a hand-crafted connector, the additional time required for the instantiation of the connector in this case has been removed and hence does not contribute to the value of T2. Thus, it can be said that the difference between X and Y are negligible.

- ⇨ (X-Y) <<< //(X-Y) is a very small value whether X>Y or Y>X
- ⇨ Equation (3) can now be written as 17125.46 = S1 – S2 --------- (4)

Value of S2 was empirically determined as an average during the execution of the experiment for heterogeneous federation. This value (246.0148 ms) when plugged in equation (4) gives

17125.46 = S1 – 246.0148

- ⇨ S1 = 16879.4452 ms

This value was then empirically verified with the execution of the Java RMI URDS instance and the client query retrieval time measured by the client of the QM. The empirical value of S1=17846 ms. Since the two values are close enough, it can be said that the higher values obtained in the case of homogeneous federation is due to the processing time of the Java-RMI URDS, i.e., higher value of S1. This is attributed to the experimental set up in which the Java-RMI URDS was deployed and executed. As mentioned in the experimental set-up, the deployment included a terminal communicating with the server hosting the file system for the Java RMI URDS; this led to a higher value of S1. Thus, it can be concluded that even though the connectors mediate between heterogeneous URDS, their effect on the time it takes for the results to

be retrieved can be small compared to the actual time taken by an individual URDS instance.

Using the calculated value of S1 and the empirically measured average value of S2, the values of X and Y can be estimated as follows:

From equation (1),

35241.73 = Z + 2(16879) + X

⇨ 35241.73 = Z + 33758 + X

⇨ X + Z = 35241.73 – 33758

⇨ X + Z = 1483.73

Let Z = constant C, since it is approximately remains the same in both the cases,

⇨ X = 1483.73 – C --------- (5)

From equation (2),

18116.27 = Z + 16879.4452 + 246.0148 + Y

⇨ 18116.27 = 17125.46 + Z + Y

⇨ Y + Z = 990.81

Again, let Z = constant C, since it is approximately remains the same in both the cases,

⇨ Y = 990.81 – C --------- (6)

Equations, (5) and (6) show that both X and Y lie within a small difference of each other's values. There have been different parameters which could have affected the values considered in the calculation (since most of the values are empirically determined) and are ignored in this analysis. Hence, the values though in close range, it cannot be determnined as to which could be greater or smaller. But since communication in both cases was binary, approximate values for X and Y confirm to be not much different from one another.

Table 6.1 Query results retrieval time measured by the DM in case of heterogeneous and homogeneous URDS federation

Heterogeneous URDSs Results Retrieval Time (ms)	Homogeneous URDSs Results Retrieval Time (ms)
18747	34420
18236	34539
19097	34960
17906	34991
17496	35081
18918	35121
17606	35181
18296	35421
17986	35752
17305	35931
17686	36262
T2 = Average Time (ms) = 18116.27	T1 = Average Time (ms) = 35241.73
S1 = 17846 ms (empirical determination for one query)	
S2 = 246.0148 ms (Average)	

This chapter studied the issue of heterogeneity and interoperability in the context of .NET and UniFrame. It helped in evaluating the UniFrame's concept of discovery service across heterogeneous component models while providing for a formal approach to achieve this. The study also establishes the principles of UniFrame's glue-generation and the approach for connector generation in achieving the federation of discovery services across two heterogeneous models - .NET and Java-RMI.

7 CONCLUSION AND FUTURE WORK

As indicated in the chapter one, the three goals of the thesis are:

1. Exploration of the .NET framework for its capability as a paradigm to build distributed applications by the integration of heterogeneous components
2. Analysis of the adaptation of the UniFrame's discovery service into .NET
3. How can the existence of multiple component models within UniFrame be handled?

This chapter concludes the thesis by providing the mechanisms that were adopted in achieving each of the above goals and the lessons learnt during the process. Section 7.1 provides a summary of the study followed by Section 7.2, which outlines the contributions of the thesis. This is followed by the future extension to the study in section 7.3. Section 7.4 ends the thesis by providing the conclusions of this study.

7.1 Summary of the Thesis

The most prominent feature of the .NET component model is its in-built support for the Web Services. The applications built with the .NET framework do not require any additional tools or wrappers for building .NET Web Services. Web Services is also the paradigm of the .NET component model, which allows the composition of heterogeneous components to function as a single distributed application. Therefore, the first objective of the thesis was achieved by analyzing this framework in particular. The analysis not only consisted of a study of the Web Services framework but also adopted a collaborative approach towards UniFrame providing for a comparison-based analysis between the two

paradigms in achieving a similar objective. The study identified metrics necessary for such a comparison and revealed that although two models can have a common objective, due to the difference between the underlying approaches there are differences in the way the process is undertaken with respect to the identified metrics. As a result, there are differences in the composed system. There are certain areas where the UniFrame approach provides for a more comprehensive solution while in the others, they both can complement each other to achieve the necessary task. For example, UniFrame does not comply with a particular interoperability mechanism to enable communication between components of heterogeneous component models. It is comprehensive enough to incorporate different mechanisms depending on the suitability of the application at hand; this includes Web Services as well (Web Services has also been discussed from the perspective of an interoperability mechanism by the thesis). Whereas, the Web Services approach is to leverage the component models such as J2EE, CORBA, etc., to comply to the Web Services framework in order to achieve the necessary interoperability. The study thus indicates that the integration environment for any distributed system must entail a detailed analysis of different choices in terms of the identified metrics.

Based on the study of the comparison metrics provided by the first objective of the thesis, the discovery service is chosen as the field of exploration for the adaptation of the UniFrame in the context of .NET. The adaptation of this part of UniFrame into .NET provides a basis for an extensive experimentation of the .NET computing model in addition to a thorough evaluation of UniFrame's adaptability. The approach used for achieving this objective was empirical in nature involving the design, prototyping and experimental evaluation of the .NET based discovery service. The prototypical realization was also compared with a similar adaptation in a different component model (Java RMI). This comparison indicated that: a) due to the differences in the underlying models of each paradigm, the creation of a prototypical discovery service required addressing many platform specific details, and b) despite these differences, the performance of both the discovery services exhibited similar behavior.

The existence of multiple component models within UniFrame is obvious and important and equally challenging problem to be tackled. The last objective of the thesis addressed the problem of heterogeneity between different component models in detail. In addition to analyzing different interoperability mechanisms, it also experimented with different commercial bridges for the .NET and Java interoperability. An architecture was proposed for interoperating between two (.NET and Java-RMI) heterogeneous discovery services, which was implemented using a specific bridge that was selected as a result of the analysis of the interoperability mechanism. The prototype was experimented with to determine the feasibility of creating a federation of heterogeneous discovery service under the UniFrame paradigm.

7.2 Contributions of the Thesis

This thesis reveals a fact that the .NET component model needs to be leveraged within a meta-component model such as UniFrame in order to achieve the realization of a DCS in a complete sense, i.e., keeping the principles of local autonomy, inherent to component computing, intact by tackling the problem of heterogeneity, which is a core feature of a DCS. This thesis provides for such an analysis for encompassing the .NET component model into UniFrame addressing different important concerns of .NET such as registration, dynamic discovery and interoperability, while utilizing the principles of UniFrame. On the other end, this thesis also provides for an evaluation of the UniFrame's meta-model approach for addressing the issue of heterogeneity and successfully incorporates the .NET component model into it. The thesis contributes towards the following main features:

- The research provided for the metrics of comparison between two system integration platforms – Web Services and UniFrame and outlined an exhaustive examination both architectural and model-based.
- Construction of a platform-specific architecture (.NET Remoting-based) from a known platform-independent URDS while providing for all the architecture mappings by successfully tackling the encountered issues. This in addition

supports an in depth evaluation of the .NET Remoting model supported by experimental results.

- Provision of an approach for tackling interoperability between different component models using the connector based technique. The approach is validated against components that are developed using different languages, operating systems, and component models. Specifically, the approach is also validated against .NET and Java RMI component models. The approach is also flexible for enabling a semi/fully automated generation of glue-wrapper code (connectors).
- Proposal of a framework for the federation of the URDS instances which spans across heterogeneous discovery services. The framework is validated with actual prototypical implementation and experimentation.

7.3 Future Work

In accordance with the goals of the thesis, the future work of the thesis can also be divided into three categories:

- Analysis of the Web Services and UniFrame paradigms.
  - The analysis currently is based on a theoretical investigation with the details provided for the identified comparison metrics. The analysis can be further strengthened by providing a case study for the construction of a DCS under the Web Services and the UniFrame paradigms. The case study can built around the same metrics as proposed as part of this thesis.
  - One possible area of collaboration proposed as a result of the analysis of this thesis, is the wrapping of components by the Web Services veneer and thereby also enriching the semantic representation of the Web Services through the domain-centric approach of the UniFrame. However, this area of collaboration needs further investigation and concretization forming a part of the future work of this thesis.

- Adaptation of UniFrame into the .NET model.
  - Adaptability of UniFrame has been studied with the adaptation of one of its constituents, namely the URDS, which included concepts such as registration and service discovery. A possible future work is to provide for such a thorough analysis for other constituents such as service descriptions and QoS validation.
  - The experimentation of the .NET URDS was in terms of performance and behavior. Though the headhunters incorporate fault-handling techniques [MYS04], it was not tested by means of experimentation, an area which needs some future work.
  - There is also a need for testing further scaling of the .NET URDS by utilizing different attribute values of the .NET Remoting model. The current experimentation worked with the default values.
- Interoperability within the context of UniFrame studied with respect to .NET.
  - Connector implementation has been tested in the thesis with commercial bridges. There is a need for a formal specification of the connectors to incorporate multiple bridging and interoperability mechanisms.
  - Design and implementation of the Glue-Wrapper Generator to utilize the above specifications and automate the generation of the connectors. The thesis references such as approach but the prototype included only the hand-crafted connector.
  - The thesis experimented .NET's interoperability with only the Java RMI component model. The examination should be extended to other component models as well.
  - The incorporated interoperability model of this thesis can be tested and applied to other entities of the UniFrame model, other than linking discovery services.

7.4 Conclusions

The thesis has provided an approach for the encompassing of the .NET component model into the UniFrame paradigm. It addresses two of the three main challenges (architecture-based interoperability and distributed resource discovery) of the UniFrame approach with a preliminary exploration of the third challenge (validation of quality requirements). The study examines the UniFrame's approach while encompassing .NET into it and also provides a mechanism for addressing the heterogeneity within the UniFrame. For the UniFrame paradigm to be able to encompass different component models, issues such as discovery, description, and integration need to be tackled in the context of these heterogeneous models. Also, it is necessary that the models, which are to be included in the UniFrame approach, are studied in a synergy with the principles of the UniFrame and the approach provided in this thesis can serve as the guideline for achieving such inclusions.

LIST OF REFERENCES

[AUG00] Auguston, M., "Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars," Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, pp. 159-166, 2000.

[BAL01] Balek, D. and F. Plasil, "Software Connectors and Their Role in Component Deployment," In Proceedings of DAIS'01, Krakow, Kluwer, September 2001.

[BAR00] Barrett R. B., "Object-Oriented Natural Language Requirements Specification," In Proceedings of ACSC 2000, The 23rd Australasian Computer Science Conference, Canberra, Australia, pp. 24-30, January 2000.

[BER03] Bertolino, A. and R. Mirandola, "Modeling and Analysis of Non-functional Propoerties in Component-based Systems," 2003, http://www1.elsevier.com/gej-ng/31/29/23/133/50/37/82.6.016.pdf.

[BRA01] Brahnmath, G., "The UniFrame Quality of Service Framework," MS Thesis, Department of Computer & Information Science, Indiana University – Purdue University Indianapolis, December 2002.

[BRI01] JNBridge for .NET-Java Interoperability, http://www.jnbridge.com/logdemo10.htm.

[BRI02] iHUB Bridge for .NET-Java Interoperability, http://www.stryon.com/products.asp?s=3.

[BRI03] iHUB Bridge, White paper, http://www.stryon.com/ihubwhitepaper.htm.

[BRI04] Ja.NET Bridging Solution for Java-.NET Interoperability, Intrinsyc, Inc., 2004, http://j-integra.intrinsyc.com/ja.net/doc/.

[BRI05] Janeva, .NET-CORBA Interoperability Bridge, http://www.borland.com/janeva/.

[BRI06] Borland's Janeva Tool for CORBA and J2EE Interoperability with .NET, http://www.borland.com/janeva/.

[BRI07] Gabhart, K., "Java/.NET Interoperability via Shared Databases and Enterprise Messaging," http://www.devx.com/interop/Article/19952.

[BUL00] Bulej, L. and B. Tomas, "A Connector Model Suitable for Automatic Generation of Connectors," http://nenya.ms.mff.cuni.cz/publications.phtml.

[CAP04] Cape Clear Integration Solution, 2004, http://www.capeclear.com.

[CLI02] Microsoft Online Clipart Gallery, 2002, http://dgl.microsoft.com/default.asp.

[COR01] OMG's Official CORBA Website, http://www.corba.org/.

[CZA00] Czarnecki, K. and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[CZE99] Czerwinski, S. E., B. Y. Zhao, T. D. Hodes, A. D. Joseph and R. H. Katz, "An Architecture for a Secure Service Discovery Service," Proceedings of Mobicom, 1999, http://ninja.cs.berkeley.edu/dist/papers/sds-mobicom.

[DIE03] Dietel, H., P. Dietel, B. DuWaldt and L. Trees, *Web Services – A Technical Introduction*, Upper Saddle River, New Jersey, Prentice Hall, 2003.

[DIS01] Coulouris, G., J. Dollimore and T. Kindberg, *Distributed Systems – Concepts and Design*, Third Edition, Addison-Wesley, 2001.

[EBX] ebXML, Enabling a Global Electronic Market, http://www.ebxml.org/.

[EMA03a] Personal e-mail communications with Mr. Naufal Khan, Vice-President – Engineering, Styron, Inc., "*The .NET interoperability-related research findings of the UniFrame team have resulted in the enhancement of our product, iHUB. This is an example of the research-industry interactions and we believe that research projects, such as UniFrame, will continue to positively affect the industrial world,"* November 2003.

[EMA03b] Personal e-mail communications with Tomas Bures, Author, "A Connector Model Suitable for Automatic Generation of Connectors," "*As to the .NET framework - we currently do not support .NET; however, generation of connectors for .NET (probably using SOAP as a transport method) sounds very interesting to me*," October, 2003.

[FRE02] Freeman J. and J. Hansome, Technical Report, "URDS Prototype," TR-CIS-1212-02, 2002.

[GER99] Geraghty, R., S. Joyce, T. Moriarty and G. Noone, "COM – CORBA Interoperability," Prentice-Hall, Inc., 1999.

[GLO04] Towards Open Grid Services Architecture (OGSA), 2004, http://www.globus.org/ogsa/.

[GOK02] Gokhale, A., D. C. Schmidt, B. Natarajan and N. Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," 2002, http://www.cse.wustl.edu/~schmidt/PDF/CACM02.pdf.

[GUP03] Gupta, N., R. Raje and A. Olson, "Analysis of the UniFrame and Web Services paradigms," SouthEast Software Engineering Conference, Huntsville, Alabama, 2003.

[GUT99a] Guttman, E., C. Perkins, J. Veizades and M. Day, "Service Location Protocol, Version 2," IETF, RFC 2608, June 1999, http://www.rfc-editor.org/rfc/rfc2608.txt.

[HUA01] Huang, Z., "The UniFrame System-Level Generative Programming Framework," MS Thesis, Department of Computer & Information Science, Indiana University – Purdue University Indianapolis, August 2003.

[HUD02] Hudson, M. J., "The Web Services Placebo," 2002, http://www.intelligententerprise.com/020917/515e_business1_1.shtml.

[ING02] Personal e-mail communication, Ingo Rammer, Author of books, *Advanced .NET Remoting* and *Advanced .NET Remoting in VB.NET,* June 2002.

[INT01] Guest, S., "Microsoft .NET and Java, Achieving Interoperability," 2001, http://www.devx.com/interop/Article/19928/1954?pf=true.

[JAV01] Java 2 Platform, Enterprise Edition, Sun Microsystems, 2001, http://java.sun.com/j2ee/index.jsp.

[MEH00] Mehta, N., R. N. Medvidovic and S. Phadke, "Towards a Taxonomy of Software Connectors," International Conference on Software Engineering, Proceedings of the 22nd International Conference on Software Engineering, 2000, ISBN:1-58113-206-9, 2000.

[MEN98] Mencl, V. "Component Definition Language," Master Thesis, Charles University, Prague, 1998.

[MIC01a] Obermeyer, P. and J. Hawkins, Microsoft Corporation, "Microsoft .NET Remoting – A Technical Overview," Official MSDN Website, 2001, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp.

[MIC01] DCOM, Microsoft Corporation, 2001, http://www.microsoft.com/com/tech/DCOM.asp.

[MIC02] .NET, Microsoft Corporation, http://www.microsoft.com/net/.

[MIC03] Rubiolo, D., J. D. Meier, E. Jezierski and A. Mackman, "Microsoft .NET Explained," http://docs.msdnaa.net/ark_new/Webfiles/WhitePapers/nxp2.doc.

[MIC04] Rammer, I., *Advanced .NET Remoting*, Apress, ISBN: 1590590252, April 2002.

[MIC05] Dhawan, P., "Performance Comparison: .NET Remoting Vs ASP.NET Web Services,"http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetarch14.asp.

[MOC87] Mockapetris, P., "Domain Names – Implementation and Specification," RFC 1035, October 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[MUK89] Mukhopadhyay, S., and M. A. L. Thathachar, "Associative Learning of Boolean Functions," IEEE Transactions on Systems, Man, and Cybernetics, 19:1008-1015, 1989.

[MYS04] Mysore P., R. R. Raje, A. M. Olson, B. R. Bryant, M. Auguston and C. Burt, Technical Report, "Scalability and fault handling issues in UniFrame Discovery Service," TR-CIS-0705-04, 2004.

[NIN02] Ninja, "The Ninja Project," 2002, http://ninja.cs.berkeley.edu.

[OMG00] Object Management Group, "Trading Object Service Specification," Object Management Group, 2000, ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf.

[OMG01a] Object Management Group (OMG), "Model Driven Architecture: A Technical Perspective," Technical Report, OMG Document No. ab/2001-02-01/04, February 2001, ftp://ftp.omg.org/pub/docs/ab/01-02-04.pdf.

[OMG01b] Object Management Group, "Naming Service Specification," Object Management Group, 2001, ftp://ftp.omg.org/pub/docs/formal/01-02-65.pdf.

[PEN01] Peng, S., "Experiments with distributed multi-agent information filtering systems," MS Thesis, Department of Computer & Information Science, Indiana University – Purdue University, Indianapolis, May 2001.

[PER92] Perry, D., E. and A. L. Wolf, "Foundations for the Study of Software Architectures," ACM SIGSOFT Software Engineering Notes, October 1992.

[PIN01] Pinkston, J., "The Ins and Outs of Integration – How EAI differs from B2B Integration," Business Integration Journal, 2001, http://www.eaijournal.com/PDF/Ins&OutsPinkston.pdf.

[RAJ00] Raje, R. R., "UMM: Unified Meta-object Model for Open Distributed Systems," Proceedings of ICA3PP 2000, 4th IEEE International Conference, Algorithms and Architecture for Parallel Processing, pp.454-465, 2000.

[RAJ01] Raje, R. R., M. Auguston, B. R. Bryant, A. Olson and C. Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration, pp. 109-119, 2001.

[RAJ02] Raje, R. R., M. Auguston, B. R. Bryant, A. Olson and C. Burt, "A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components," Technical Report, Department of Computer and Information Science, Indiana University – Purdue University Indianapolis, 2002.

.

[RAJ03] Olson, M. A., R. R. Raje, B. R. Bryant, M. Auguston and C. Burt, "UniFrame, A Unified Framework for Developing Service-oriented, Component-based Distributed Software Systems," in Service-Oriented Software System Engineering: Challenges and Practice, Idea Group, Inc, 2004 (In Press).

[RAP01] Raptis, K., D. Spinellis and S. Katsikas, "Multi-Technology Distributed Objects and Their Integration," Computer Standards & Interfaces, pp.157-168, 2001.

[REM04] .NET Framework Developer's Guide – .NET Remoting Architecture, 2004, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingarchitecture.asp.

[SHA96] Shaw, M. and D. Garlan, *Software Architecture: Perspectives on Emerging Discipline*, Prentice-Hall, 1996.

[SIR01] Siram, N., "An Architecture for the UniFrame Resource Discovery Service," MS Thesis. Indiana University – Purdue University Indianapolis, March 2002.

[SKO03] Skonnard, A., Microsoft MSDN, "Understanding XML Schema," 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnxml/html/understandxsd.asp.

[STR04] Stryon Inc., 2004, http://www.stryon.com.

[SUN01a] Sun Microsystems, "Jini Architecture Specification, Version 1.2," Sun Microsystems, December 2001, http://www.sun.com/jini/.

[SUN03] Changlin, S., "QOS Composition and Decomposition in UniFrame," M. S. Thesis, Department of Computer & Information Science, Indiana University – Purdue University Indianapolis, June 2003.

[THA85] Thathachar, M. A. L. and P. S. Sastry, "A new approach to the design of reinforcement schemes for learning automata," IEEE Transactions on Systems, Man, and Cybernetics, 15:168-175, 1985.

[THU01] Thuan, T. and H. Lam, *.NET Framework Essentials*, Chapter 6, Web Services, 2001, http://www.oreilly.com/catalog/dotnetfrmess/chapter/ch06.html.

[VAN65] Van A., "Orthogonal Design and Description of a Formal Language," Technical report, Mathematisch Centrum, Amsterdam, 1965.

[WAH97] Wahl, M., T. Howes and S. Kille, "Lightweight Directory Access Protocol (v3)," IETF RFC 2251, December 1997, http://www.rfc-editor.org/rfc/rfc2251.txt.

[WEB01] The Web Services Community Portal, 2004, http://www.webServices.org.

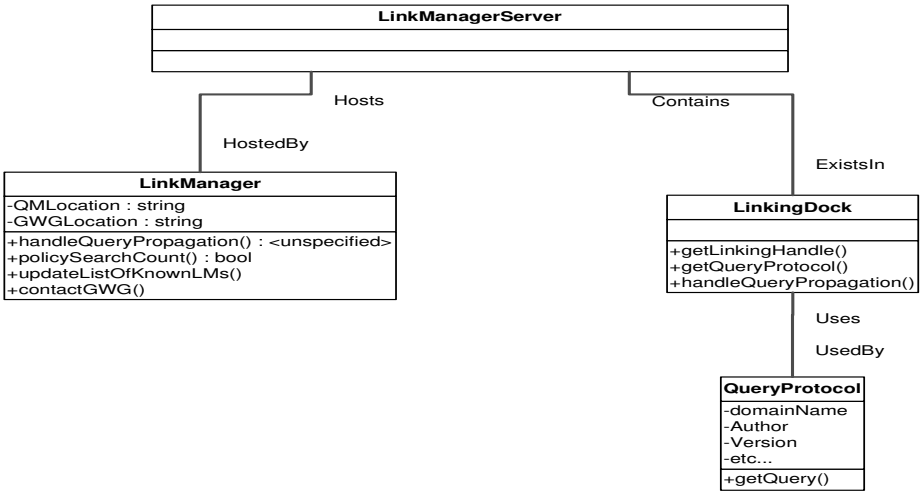
[WEB02] Samtani, G. and D. Sadhwani, “Web Services and Application Frameworks Working Together,” 2002, http://www.webservicesarchitect.com/content/articles/samtani04print.asp.

[WEB03] Building Interoperable Web Service: WS-I Basic Profile 1.0, 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsvcinter/html/wsi-bp_chapter1.asp.

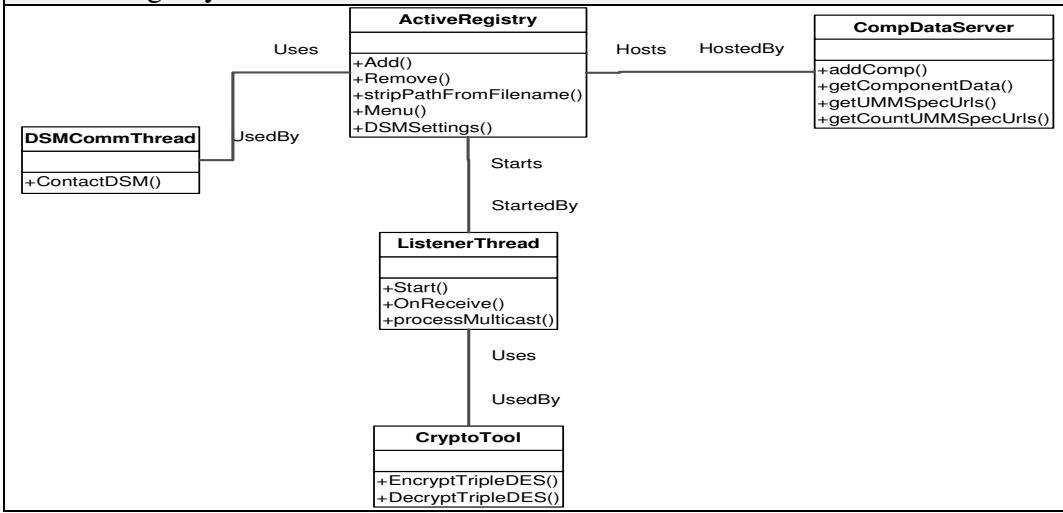
APPENDIX

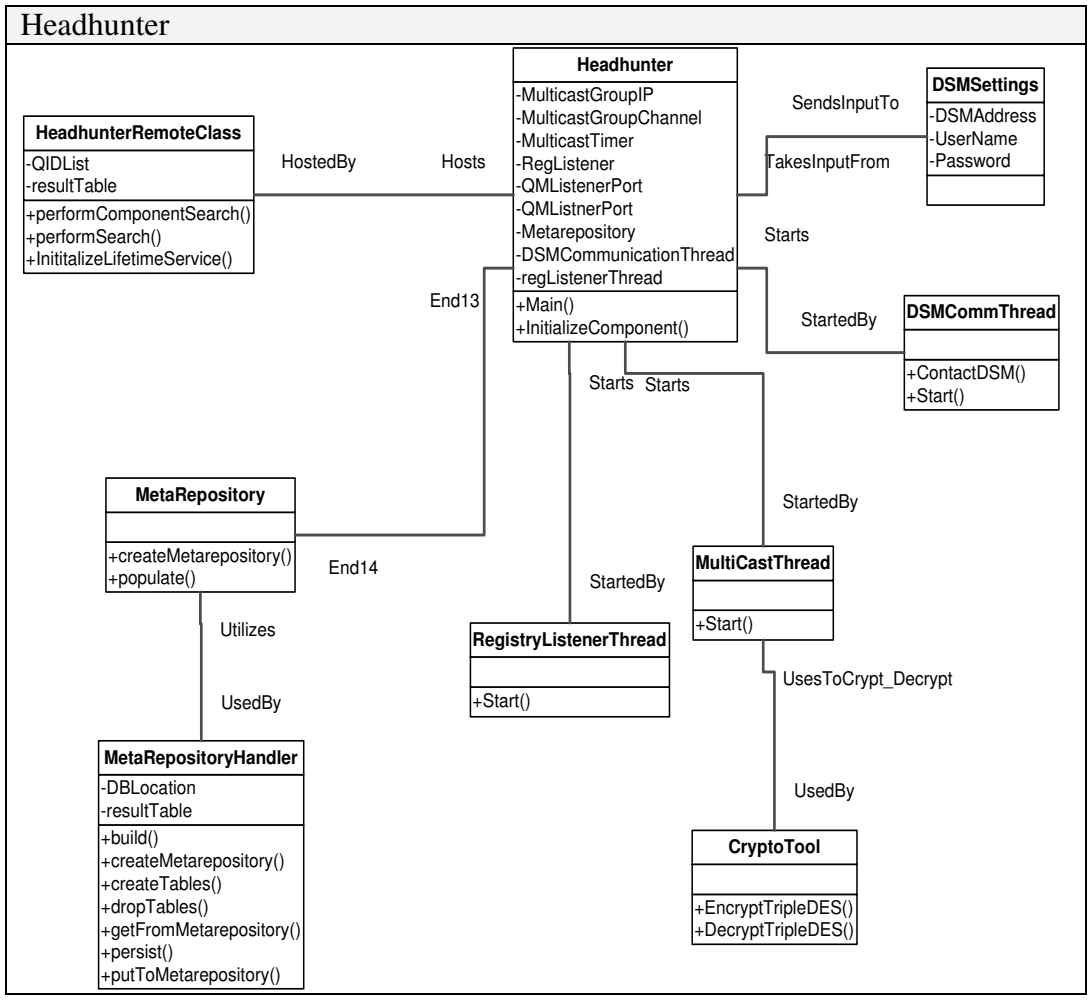
CLASS DIAGRAMS

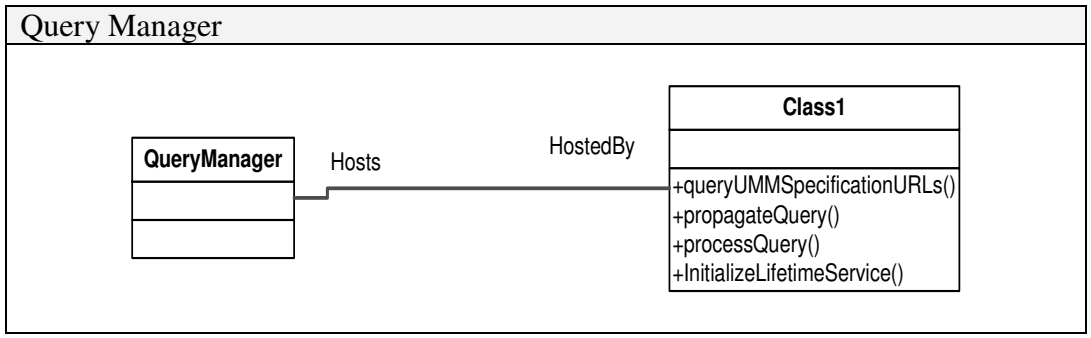
Link Manager



Active Registry




Headhunter
Headhunter
-MulticastGroupIP
-MulticastGroupChannel
-MulticastTimer
-RegListener
-QMListenerPort
-QMListnerPort
-Metarepository
-DSMCommunicationThread
-regListenerThread
+Main()
+InitializeComponent()
HeadhunterRemoteClass
-QIDList
-resultTable
+performComponentSearch()
+performSearch()
+InititalizeLifetimeService()
HostedBy
Hosts
SendsInputTo
TakesInputFrom
DSMSettings
-DSMAddress
-UserName
-Password
Starts
StartedBy
DSMCommThread
+ContactDSM()
+Start()
End13
Starts
Starts
MetaRepository
+createMetarepository()
+populate()
End14
StartedBy
MultiCastThread
+Start()
StartedBy
RegistryListenerThread
+Start()
Utilizes
UsedBy
UsesToCrypt_Decrypt
MetaRepositoryHandler
-DBLocation
-resultTable
+build()
+createMetarepository()
+createTables()
+dropTables()
+getFromMetarepository()
+persist()
+putToMetarepository()
UsedBy
CryptoTool
+EncryptTripleDES()
+DecryptTripleDES()


Query Manager
QueryManager
Hosts
HostedBy
Class1
+queryUMMSpecificationURLs()
+propagateQuery()
+processQuery()
+InitializeLifetimeService()

Domain Security Manager